SEL-89-007

PROCEEDINGS OF THE FOURTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

November 29, 1989

GODDARD SPACE FLIGHT CENTER

Greenbelt, Maryland

FOREWORD

The Software Engineering Laboratory (SEL) is an organization sponsored by the National Aeronautics and Space Administration/Goddard Space Flight Center (NASA/GSFC) and created for the purpose of investigating the effectiveness of software engineering technologies when applied to the development of applications software. The SEL was created in 1977 and has three primary organizational members:

NASA/GSFC, Systems Development Branch

The University of Maryland, Computer Sciences Department

Computer Sciences Corporation, Systems Development Operation

The goals of the SEL are (1) to understand the software development process in the GSFC environment; (2) to measure the effect of various methodologies, tools, and models on this process; and (3) to identify and then to apply successful development practices. The activities, findings, and recommendations of the SEL are recorded in the Software Engineering Laboratory Series, a continuing series of reports that includes this document.

Single copies of this document can be obtained by writing to

Systems Development Branch
Code 552
Goddard Space Flight Center
Greenbelt, Maryland 20771

AGENDA

FOURTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP
NASA/GODDARD SPACE FLIGHT CENTER
BUILDING 8 AUDITORIUM
NOVEMBER 29, 1989

Summary of Presentations
R. W. Kester (CSC)

Session 1

Topic: Studies and Experiments in the SEL

The Experience Factory: Packaging Software Experience
V.R. Basili (University of Maryland)

Experiences in the SEL – Applying Software Measurement
F.E. McGarry (NASA/GSFC)
S.R. Waligora and T.P. McDermott (CSC)

Evaluation of the Cleanroom Methodology in the SEL
A. Kouchakdjian and V.R. Basili (University of Maryland)
S. Green (NASA/GSFC)

Session 2

Topic: Methodologies

Predicting Project Success from the Software Project Management Process: An Exploratory Analysis
M.S. Deutsch (Hughes Aircraft Co.)

A Software Environment: Some Surprising Empirical Results
B.I. Blum (APL)

Measurement Based Improvements of Maintenance in the SEL
H.D. Rombach and B.T. Ulery (University of Maryland)
J.D. Valett (NASA/GSFC)

AGENDA (Cont'd)

Session 3

Topic: Software Reuse

Software, System, and Application Uncertainty and Its Control Through the Engineering of Software
M. Lehman (Imperial College)

Testing in a Reuse Environment – Issues and Approaches
J.C. Knight (University of Virginia)

Domain-Directed Reuse
C. Braun and R. Prieta-Diaz (Contel)

Using Reverse Engineering and Hypertext to Document an Ada Language System
K. Thackrey (Planning Analysis Corporation)

Session 4

Topic: Testing and Error Analysis

Classification Tree Analysis Using the Amadeus Measurement and Empirical Analysis System
R.W. Selby, G. James, K. Madsen, J. Mahoney, A.A. Porter, and D.C. Schmidt (U. C. Irvine)

The Jet Propulsion Laboratory's Experiences with Formal Inspections
M. Bush and J. Kelly (JPL)

The Enhanced Condition Table Methodology for Verification of Fault Tolerant and Other Critical Software
M. Hecht, K.S. Tso, and S. Hochhauser (SoHaR, Inc.)

Appendix A – Attendees

Appendix B – Standard Bibliography of SEL Literature

SUMMARY OF PRESENTATIONS

Rush Kester, Computer Sciences Corporation

SUMMARY OF THE FOURTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP

On November 29, 1989, approximately 450 attendees gathered in Building 8 at the National Aeronautics and Space Administration (NASA)/Goddard Space Flight Center (GSFC) for the Fourteenth Annual Software Engineering Workshop. The meeting is held each year as a forum for information exchange in the measurement, utilization, and evaluation of software methods, models, and tools. It is sponsored by the Software Engineering Laboratory (SEL), a cooperative effort of NASA/GSFC, Computer Sciences Corporation (CSC) and the University of Maryland. Among the audience were representatives from 10 universities, 22 government agencies, 9 NASA centers, and 83 private corporations and institutions. Thirteen papers were presented in four sessions:

- Studies and Experiments in the SEL
- Methodologies
- Software Reuse
- Testing and Error Analysis

SESSION 1 - STUDIES AND EXPERIMENTS IN THE SEL

Frank McGarry of GSFC opened the workshop, welcomed attendees, and introduced the first speaker. The first presentation "Packaging Experience for an Improved Process" was given by Victor Basili of the University of Maryland. Basili indicated that a major purpose of the SEL has been evaluating different technologies and methods of software development and providing feedback to project managers for improving the process.

SEL studies have been guided by the Goal/Question/Metric paradigm and its corollary, the Improvement paradigm. Each study determines how best to package experiences for most effective reuse. Reuse of experiences, to date, have generally been adhoc and informal. However, to maximize the benefit to the organization, more formality is needed. One problem has been that the goal of projects is to produce their own items, not to capture, generalize, and communicate experiences to other projects for reuse.

Part of the solution, Basili stated, is the creation of an organization, "the experience factory," whose goal is to facilitate transferring experience and products from producer projects to reuser projects. The raw materials for this factory are plans, models, products, status, and lessons learned collected throughout each project's life cycle. These inputs are processed by the factory and, as appropriate, stored in an experience base or discarded. The experience factory produces feedback of comparative project status and reusable items from the experience base. Where it is not possible to provide automated support, the factory can provide consulting services.

Basili believes almost every type of experience can be packaged for reuse. One major issue is how the "experience factory" should be funded. Whether treated as overhead or a cost center, over time it should pay for itself within the organization by improvements in the process or products. Basili ended with the good news that an organization can start small and expand its experience factory as managers understand how best to serve the organization.

The second speaker, Frank McGarry, presented "Experiences in the SEL Applying Software Measurement." The projects developed (in the Flight Dynamics environment) are medium sized (80K to 100K source lines of code (SLOC)) and average 2 years in duration. Most development is in FORTRAN with 10-15 percent in Ada. Over the past 14 years, the SEL has collected data including cost, error, product, methodology, and tools on over 75 projects. There are four areas where the SEL applies the data collected: understanding the development environment, managing current projects, planning future projects, and providing rationale for adopting standards and methodology.

McGarry stated that the first application of data collection, understanding the environment, helps the organization identify its strengths and weaknesses and start building models of the development process. For example, the SEL has found that the distribution of effort according to milestones differs somewhat between FORTRAN and Ada. However, there is little difference in distribution of effort by type of activity. This similarity is due to the significance of environmental factors that change very slowly.

McGarry used the error model as an example of the second application of data collection for managing current projects. By monitoring errors during the code and test phase compared to prior projects, a manager can determine whether the project's performance was typical or required remedial management actions. Measurement of computer utilization can also be applied to help manage projects. For instance, a manager observing abnormally low usage could upon further investigation uncover problems with lack of requirements definition or resource availability.

The third application of data collection is for planning future projects. Without data collected from prior projects, an organization cannot make plans that reflect its way of doing business. Using models developed from historical data, the SEL can, given an estimate of project size, predict the effort or cost for the project and its allocation to the phases of the life cycle. In addition, collected data have been used to develop other rules-of-thumb relations among various measures.

The fourth application of data collection McGarry described is to provide rationale for adopting new methodologies or standards. This Improvement paradigm closes the loop on the measurement process. For example, measurement of software reuse during experiments with Object-Oriented Development (OOD) has lead the SEL to incorporate OOD in its development methodology. Data collected during other experiments has enabled the SEL to put aside unsuccessful tools or methods. In wrapping up, McGarry pointed to the measurable improvements in the SEL's software productivity and reliability over time as evidence of the benefit of data collection in the evolution of standards and methodology.

The final speaker in the first session, Ara Kouchakdjian from the University of Maryland, presented "Evaluation of the Cleanroom Methodology in the SEL." The Cleanroom method was conceived at IBM with a goal of producing correct code the first time. The emphasis is on the use of human evaluation rather than computer debugging to verify software. The Cleanroom discipline is characterized by the complete separation of coders and testers. The importance of correctness is emphasized by not allowing development to proceed from design to code or from code to test until all reviewers were convinced of the product's correctness.

Kouchakdjian described the project used in the Cleanroom experiment as a 33 thousand SLOC production subsystem. The project was staffed by five individuals spending about half-time on the project. None of the team had prior experience using the Cleanroom methodology or on this specific type of application. Following 1 month of Cleanroom training, the project has taken 22 months and is currently completing system test. The effort that remains is integration with the rest of the system and acceptance testing.

The Cleanroom project was compared to the typical SEL project by Kouchakdjian. The Cleanroom project spent 10 percent more of its total effort in design and 2-3 percent less in code and test than the typical SEL project. During coding the Cleanroom project spent 52 percent of its time reading code versus 15 percent for typical SEL projects. The error rate for the Cleanroom project was 2.7 per 1000 SLOC versus 6.0 for typical projects. Of the Cleanroom project's errors, 33 percent were found during code reviews and 54 percent during code reading. The productivity of the Cleanroom project was 4.9 SLOC per staff-hour versus 2.9 typical in the SEL environment. From these results, Kouchakdjian concluded that the Cleanroom methodology appears promising, but further work is needed.

SESSION 2 - METHODOLOGIES

Michael Deutsch of Hughes Aircraft presented "Predicting Project Success from the Software Project Management Process: An Exploratory Analysis." The goal of this study was to identify, empirically, the project management factors that most strongly correlate with project success and those factors that best discriminate between success and failure. This study proposed a hypothetical model of project success in which project adversity factors such as size, interfaces, business, and technical constraints combine with management power factors such as resources, scope definition, risk management, planning, and user/customer/contractor dialogue to form "Net Turbulence." This "Net Turbulence" parameter determines a project's business and/or technical performance.

In the study, an informal questionnaire was given to available project managers and senior engineers on 25 completed projects. The projects ranged in size from 25 thousand to 2 million SLOC. The study found that the overall perception of project success was based on business rather than technical performance, with a threshold between perceived success and failure being a 25 to 50 percent overrun and a 3 to 6 month schedule slip. Deutsch asserted that the driving factor in determining the degree of project success is the degree to which the user/customer/

contractor dialogue produced a mutual agreement that the right problem was being solved.

Deutsch indicated that factors most highly correlated with project success confirmed management theory and anecdotes. One surprising finding was the strength of "engineering and application expertise of the initial maintenance team" in determining the success of all projects. This factor ranked first in its correlation with Business Performance and second in its correlation with Technical Performance. Deutsch closed with an example that pointed out the potential practical value of the "Net Turbulence" model. The model identified adversity factors that by themselves might lead to project failure but, when coupled with application of appropriate management power factors, often lead to project success.

The second speaker of this session, Bruce Blum of the Applied Physics Laboratory, presented "A Software Environment, Some Surprising Empirical Results." Blum presented observations of information systems development using a program generator as indicative of how the software process might behave if programming were eliminated. In this environment, systems would be developed by users, or a small staff of applications experts, and would continually evolve along with user needs. The primary system used in the study was the clinical information system used for cancer treatment at Johns Hopkins University. The size of the system in 1988 was 6600 programs and 1600 tables, containing 600,000 patient-days of data.

Looking at the growth of the system Blum found it fairly steady, whether the system was newly developed or mature. This was due to the insatiable nature of users. After 5 years of use, one-third of the programs and tables are new. By comparison, while one-third of the programs had been edited, only 7 percent of the tables required editing, indicating greater stability in the data model. Even with this large number of changes, only a small maintenance team was required. Very little computer experience was needed; rather, the individuals became domain experts through on-the-job training. Blum summarized by noting that with the difficulty of system implementation removed, inherent individual differences became less important to productivity and that the system became more integrated.

The last speaker of the morning sessions, H. Dieter Rombach from the University of Maryland, presented "Measurement Based Improvement of Maintenance in the

SEL." The goals of this study were to understand and characterize early maintenance and, where possible, provide feedback to improve the maintenance and development processes. This effort studied six satellite attitude systems developed in FORTRAN using the standard SEL methodology. The systems ranged in size from 37 to 235 thousand SLOC and their development efforts from 3 to 28 staff-years. The data used in this study were collected from weekly activity reports, change reports, and subjective interviews with maintenance personnel.

In analyzing the types of requests for software changes, Rombach found that while 53 percent were for error corrections, this represented only 27 percent of the effort (slightly less than one-half that required per adaptation or enhancement request). The study found no obvious correlation whereby the maintenance effort could be predicted from the development effort or system size. Not surprisingly, the study found that changes during maintenance required more time than changes during development. However, it was surprising to find that this increase was due more to increased effort to implement and integrate the change than to increased effort to isolate the problem.

Based on interviews with maintenance personnel, Rombach found that the subject software was poorly suited to maintenance needs in the following ways: (1) program design language (PDL) is redundant with code and inconsistencies just added confusion, (2) specification of the same information in multiple locations leads to incomplete changes, and (3) debug output of the form "variable = value" requires too much familiarity with the code. In closing, Rombach indicated that future studies will focus on extended maintenance data for these systems and early maintenance of Ada systems.

SESSION 3 - SOFTWARE REUSE

In a more philosophical vein, Manny Lehman of Imperial College presented "Uncertainty in Computer Applications." Computer programs can be classified as one of three types: (1) those completely defined by a specification, (2) those whose solution need not be exact but merely close enough for a specific problem, and (3) those that fulfill an application in the real world and whose success is based on user satisfaction. This third type of program was the focus of this talk.

Lehman stated that real world applications continually evolve because the real world changes and the user's needs change. Software maintenance is the means of achieving this evolution. What is maintained is the level of user satisfaction and the validity of assumptions embedded in the program. Lehman estimated that one assumption about the real world is embodied in every 10 SLOC. Some of these assumptions were probably questionable from the start while others were initially valid but become invalid over time. As a result, execution of real world applications involves some uncertainty and risk.

Minimizing the risk due to the presence of uncertain assumptions, Lehman concluded, is a professional responsibility. To accomplish this, the software process must (1) carefully document all assumptions (explicit and implicit) and (2) periodically review assumptions to ensure they continue to be correct and appropriate.

Next John Knight, from the University of Virginia, presented "Testing in a Reuse Environment - Issues and Approaches." Testing of parts for a reuse library presents some unique challenges. Testing a part for one application and testing for every possible application is significantly different. This additional testing is justified, provided its cost can be amortized by future instances of reuse.

Knight presented several interesting approaches to testing adaptable parts, such as Ada generics. Where the design of the part restricts the allowable range of parameters or relations between parameters, these restrictions can be validated by the code itself. Where a broad range of parameters is allowed, a program generator could be created to test the adaptable part across this domain. Where generic parameters are executable subprograms that must provide specific semantics, validation cannot generally be automated. Rather, the designer of the adaptable part should write a specification for the subprogram and define a test procedure for its validation.

In conclusion, Knight stated that while reuse can have a significant impact on testing, it doesn't make testing any easier, as some economic models assume. However, if one is careful, perhaps the total resources expended can be reduced.

Chris Braun, from Contel, spoke next on "Domain-Directed Reuse." Domain-directed reuse is an approach that combines top-down generative reuse with bottom-up compositional reuse. Generative reuse is an approach whereby systems are generated automatically by specifying a set of parameters that tailor a given architecture, e.g., a program generator. Compositional reuse occurs when components are selected from a library and used to build a system.

The system envisioned by Braun is one that presents the user with a graphic representation of a standard architecture for a given application domain, e.g., Command, Control, and Communications. For each level in the architecture, its building blocks are represented. Where components exist for a given building block, the user may select one from the component library. Where no appropriate component exists, the user builds one from scratch or by assembling suitable existing components from lower levels. This new component must conform to the interface and functional requirements required by the standard architecture. In this way, the user would be guided through the design process. Braun concluded by predicting significant long-term gains in effective productivity for system development utilizing domain-directed reuse.

The final speaker of this session, Kent Thackrey from Planning Analysis Corporation, presented "Using Reverse Engineering and Hypertext to Document an Ada Language System." When asked to document an existing system (650 modules, 40,000 SLOC), rather than deliver an estimated 2500 pages of documentation, this project developed interactive documentation using hypertext. Users could traverse the system, moving up or down the call tree, viewing module descriptions. By pressing a special key, the module's source code was displayed. If the module generated a screen or accessed a file, the screen layout or record descriptions could be viewed by pressing other special keys.

Thackrey estimated that 60-70 percent of the documentation was automatically generated by parsing the source. The remaining information was derived by manually reverse engineering the system. The hypertext documentation met the customer's standards and was well-received and heavily used by the maintenance personnel. However, the hypertext documentation has not been maintained along with the system and is becoming less useful, a situation that could be remedied by better training and automated procedures for maintaining the hypertext documentation. Thackrey closed by describing enhancements to the hypertext documentation structure that he felt would extend its usefulness not only for system maintenance but for navigation of a reuse library.

SESSION 4 - TESTING AND ERROR ANALYSIS

Richard Selby, from the University of California at Irvine, presented "Classification Tree Analysis Using the Amadeus Measurement and Empirical Analysis System." The Amadeus System, which is currently being prototyped, provides the conceptual framework for instrumenting a software development environment. A software development environment that provides the required Amadeus interfaces will allow automatic measurement and monitoring of the development process or its objects. The focus of this presentation was Selby's experience using metrics-based classification trees.

The goal of classification tree analysis, Selby stated, is to identify automatically that small portion of a system's components that is likely to account for a disproportionately high amount of its cost, and, thus, focus management attention on development resources. Classification trees can be defined using any combination of nominal, ordinal, interval, or ratio metrics. Software to assist in the generation of classification trees from empirical data was studied in trials at GSFC and Hughes. Selby concluded that these proof of concept studies, which sought to identify high-cost and error-prone modules, have demonstrated that the automatic generation of classification trees has merit.

The next speaker, Marilyn Bush of the Jet Propulsion Laboratory (JPL), presented "The Jet Propulsion Laboratory's Experiences with Formal Inspections." Formal inspections at JPL are based on the technique published by Michael Feigan of IBM in 1976. Inspections are designed to find, document, fix, and verify defects as early in the life cycle as possible. At JPL inspections span the life cycle, starting with system requirements through test procedures. The inspection process is the same throughout and includes preparation, overview, the actual inspection meeting, rework, and verification.

Bush described inspections as lead by a trained moderator with 3 to 6 peer inspectors who have a vested interest in the product. Each inspector completes a well-defined checklist specific to the product and phase before the inspection meeting. The material for an inspection is limited (about 40 pages or 600 SLOC) so that the meeting lasts no more than 2 hours. The average inspection at JPL consumed about 28 staff-hours and found 16 defects, including 4 defects that would have prevented the system from operating correctly. Bush estimated that each inspection saved JPL about $25,000.

Bush credited training as essential to the success of inspections at JPL. JPL developed a 2-hour course for managers that stressed the value of inspections and a 1.5-day course for inspectors and moderators that described the inspection process and its benefits, and included a simulated inspection. In addition, support and consultation were provided to projects during their initial use of inspections. Bush concluded by citing some lessons learned.

The final presentation, "The Enhanced Condition Table Methodology for Verification of Fault Tolerant and other Critical Software," was given by Myron Hecht of SoHaR, Inc. This technique is based on a test data selection method described by Goodenough and Gerhart in 1975. The technique is expensive and is justified only where severe reliability requirements exist, e.g., critical modules in a nuclear reactor control system. The original technique, which was impractical for programs over 20 lines, was enhanced in two ways. First, tools were created that automated condition table generation and assisted in eliminating don't care test cases. Second, the method integrated structural testing with analysis of functional, reliability, and safety requirements.

Hecht described an experiment using the module responsible for the node manager functions in a highly redundant network. The module was about 150 SLOC and contained 14 conditions. Without the enhanced method, this would have resulted in 2**14 combinations. The enhanced condition table contained 50 combinations. The technique identified one error of omission that Hecht believes would not have been identified by any other method. To emphasize his belief that no testing technique by itself is sufficient, Hecht indicated that 14 additional interface and timing errors were identified by functional testing during integration.

Hecht closed by stating that when additional tools are developed to reduce the labor and time intensive tasks of creating test cases and test environments, this technique will be a thorough, traceable, and effective means of performing unit testing where certification is required.

SESSION 1 — STUDIES AND EXPERIMENTS IN THE SEL

V. R. Basili, University of Maryland

F. E. McGarry, NASA/GSFC

A. Kouchakdjian, University of Maryland

The Experience Factory: Packaging Software Experience

Victor R. Basili
Institute for Advanced Computer Studies
and
Department of Computer Science
University of Maryland

In order to improve software quality and productivity, we need to build descriptive models to better understand (1) the nature of the processes and products and their various characteristics, (2) the variations among them, (3) the weaknesses and strengths of both, and (4) mechanisms to predict and control them. Based upon analysis of these descriptive models, we need to build prescriptive models that improve both the products and the methods for developing them relative to a variety of qualities, provide feedback for project control, and allow the packaging of successful experience. We also need to examine the interaction among these models.

The overall solutions are technical and managerial. The technologies involved include modeling, measurement and reuse.

We have been applying this basic approach for the past 14 years at NASA/GSFC in a program called the Software Engineering Laboratory (SEL). The activities were broken into two phases. During the first phase, we worked to understand the environment and how to measure it. To achieve this we measured what we could, applied whatever models existed, built baselines for such things as defect classes and resource allocation, and developed the Goal/Question/Metric paradigm as an organized mechanism for setting goals and measuring the software process and product.

During this phase, we learned that although there are similarities among software developments, the differences are what create the problems; that there is a direct relationship between the processes performed and the various product qualities; that measurement needs to be based upon goals and models; and that evaluation and feedback are necessary for project control.

In phase two we worked to improve the process and product quantitatively based upon the evolutionary development of various models. To this end, we experimented with technologies, evaluated and fed back information to the project, developed the Quality Improvement Paradigm which is a variation of the scientific method tailored to the software domain, began formalizing process, product, knowledge and quality models for the environment, and continued to evolve the GQM paradigm and our various models.

During this phase, we learned that evaluation and feedback are necessary for learning; that process, product and quality models need to be more formally defined and tailored for the particular environment; that software development should follow an experimental approach; that reusing experience in the form of process, product, and knowledge is essential; and that experience needs to be packaged.

The Improvement Paradigm consists of six steps:

1. Characterize the current project environment.

2. Set up goals and refine them into quantifiable questions and metrics for successful project performance and improvement over previous project performances.

3. Choose the appropriate software project execution model for this project and supporting methods and tools.

4. Execute the chosen processes and construct the products, collect the prescribed data, validate it, and analyze the data to provide feedback in real-time for corrective action on the current project.

5. Analyze the data to evaluate the current practices, determine problems, record the findings and make recommendations for improvement for future projects.

6. Package the experience in the form of updated and refined models and other forms of structured knowledge gained from this and previous projects, and save it in an experience base so it can be available to the next project.

The Improvement Paradigm necessitates support for systematic learning and reuse. Systematic learning requires support for the recording of experience, off-line generalizing or tailoring of experience, and the formalization of experience. Systematic reuse requires support for using existing experience and on-line generalizing and tailoring of candidate experience. Both learning and reuse need to be integrated into an overall evolution model that supports them as formal activities.

Reuse has been an elusive for software development. This is due to a number of factors. First, reuse needs to be defined as more than just the code level; emphasis on code only limits the context of reuse. Our model covers the reuse of all forms of experience, e.g., all forms of products and processes. In the past, reuse of experience has been too informal and not fully incorporated into the software evolution model. It has been assumed that reuse means using as is. Actually, most experience needs to be modified in some way. There need to be support mechanisms for this modification process. To make reuse easier, experience needs to be packaged. It also needs to be analyzed for its potential for reuse before being offered as reusable. Lastly, the development and packaging of reusable experience was expected to take place as part of the project development. Clearly, this is very difficult since the project focus is delivery, not reuse.

For these reasons, we propose the concept of an Experience Factory, which is distinct from the project organization in that it packages experience by building informal, formal or schematized, and productized models and measures of various software processes, products, and other forms of knowledge via people, documents, and automated support. As such, the Experience Factory supports project development by analyzing and synthesizing all kinds of experience, acting as a repository of such experience, and supplying that experience to projects on demand.

The Experience Factory is a logically and/or physically separate organization for the project development organization. This is necessary because the Experience Factory and Project Organizations have different focuses and priorities, and require different process models and expertise requirements.

There are a variety of different experiences that can be packaged. These consist of process models, the results of method and techniques evaluation, resource baselines and models, change and defect baselines and models, product baselines and models, and products and product parts themselves.

The benefits of the concept of an Experience Factory are: the separation of concerns from project development, the support for learning and reuse, the generation of a tangible corporate asset, and the formalization of management and development technologies. To build an Experience Factory, an organization can start small by packaging those things it knows well and building via measurement and models to larger bodies of knowledge. The concept of an Experience Factory allows us to focus research on the understanding and packaging of those pieces of experience that will aid projects the most.

Aside from the packaging of experience, the Experience Factory can incorporate other activities such as quality assurance, education and training, and consulting activities. Funding for the Factory should be a separate cost center and can come from corporate overhead or projects can be billed for packages of experience.

In conclusion, combining the concepts of the Improvement Paradigm, the Goal/Question/Metric Paradigm, and the Experience Factory organization provides a framework for software engineering development, maintenance, and research that supports the improvement of quality and productivity in an organized way. It takes advantage of the experimental nature of software engineering and allows us to understand how software is built and focused on the problems, define and formalize models of process and product with respect to success criteria, and feed back packaged experience to current and future projects for reuse.

VIEWGRAPH MATERIALS

FOR THE

V. BASILI PRESENTATION

THE EXPERIENCE FACTORY:
PACKAGING SOFTWARE EXPERIENCE

VICTOR R. BASILI
UMIACS
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF MARYLAND

HOW DO WE IMPROVE SOFTWARE QUALITY AND PRODUCTIVITY?

WE NEED TO

- UNDERSTAND PROCESS AND PRODUCT
- DEFINE PROCESS AND PRODUCT QUALITIES
- EVALUATE SUCCESSES AND FAILURES
- FEEDBACK FOR PROJECT CONTROL
- PACKAGE SUCCESSFUL EXPERIENCES

KEY TECHNOLOGIES:

- MODELING
- MEASUREMENT
- REUSE

WE NEED TECHNICAL AND MANAGERIAL SOLUTIONS

SEL EVOLUTION

EVOLVING FOR OVER 14 YEARS

PHASE I

UNDERSTAND WHAT WE COULD ABOUT THE ENVIRONMENT AND MEASUREMENT

MEASURED WHAT WE COULD

USED MODELS WHERE AVAILABLE

BUILT BASELINES AND MODELS

DEVELOPED THE GOAL/QUESTION/METRIC PARADIGM

LEARNED

- THERE ARE FACTORS THAT CREATE SIMILARITIES AND DIFFERENCES AMONG PROJECTS
- THERE IS A DIRECT RELATIONSHIP BETWEEN PROCESS AND PRODUCT
- MEASUREMENT NEEDS TO BE BASED ON GOALS AND MODELS
- EVALUATION AND FEEDBACK ARE NECESSARY FOR PROJECT CONTROL

SEL EVOLUTION

PHASE II

IMPROVE THE PROCESS AND PRODUCT

- EXPERIMENTED WITH TECHNOLOGIES
- EVALUATED AND FED BACK INFORMATION TO THE PROJECT
- DEVELOPED THE IMPROVEMENT PARADIGM
- BEGAN FORMALIZING PROCESS, PRODUCT, KNOWLEDGE, AND QUALITY MODELS
- EVOLVED THE GOAL/QUESTION/METRIC PARADIGM

LEARNED

- EVALUATION AND FEEDBACK ARE NECESSARY FOR LEARNING
- PROCESS, PRODUCT AND QUALITY MODELS NEED TO BE BETTER DEFINED AND TAILORED
- SOFTWARE DEVELOPMENT SHOULD FOLLOW AN EXPERIMENTAL APPROACH
- REUSING EXPERIENCE IN THE FORM OF PROCESS, PRODUCT, AND KNOWLEDGE IS ESSENTIAL
- EXPERIENCE NEEDS TO BE PACKAGED

IMPROVEMENT PARADIGM

1. CHARACTERIZE THE CURRENT PROJECT ENVIRONMENT.

2. SET UP GOALS AND REFINE THEM INTO QUANTIFIABLE QUESTIONS AND METRICS FOR SUCCESSFUL PROJECT PERFORMANCE AND IMPROVEMENT OVER PREVIOUS PROJECT PERFORMANCES.

3. CHOOSE THE APPROPRIATE SOFTWARE PROJECT EXECUTION MODEL FOR THIS PROJECT AND SUPPORTING METHODS AND TOOLS.

4. EXECUTE THE CHOSEN PROCESSES AND CONSTRUCT THE PRODUCTS, COLLECT THE PRESCRIBED DATA, VALIDATE IT, AND ANALYZE THE DATA TO PROVIDE FEEDBACK IN REAL-TIME FOR CORRECTIVE ACTION ON THE CURRENT PROJECT.

5. ANALYZE THE DATA TO EVALUATE THE CURRENT PRACTICES, DETERMINE PROBLEMS, RECORD THE FINDINGS AND MAKE RECOMMENDATIONS FOR IMPROVEMENT FOR FUTURE PROJECTS.

6. PACKAGE THE EXPERIENCE IN THE FORM OF UPDATED AND REFINED MODELS AND OTHER FORMS OF STRUCTURED KNOWLEDGE GAINED FROM THIS AND PREVIOUS PROJECTS, AND SAVE IT IN AN EXPERIENCE BASE SO IT CAN BE AVAILABLE TO THE NEXT PROJECT.

SYSTEMATIC LEARNING AND REUSE

SYSTEMATIC LEARNING REQUIRES SUPPORT FOR

- RECORDING EXPERIENCE
- OFF-LINE GENERALIZING OR TAILORING OF EXPERIENCE
- FORMALIZING OF EXPERIENCE

SYSTEMATIC REUSE REQUIRES SUPPORT FOR

- USING EXISTING EXPERIENCE
- ON-LINE GENERALIZING OR TAILORING OF CANDIDATE EXPERIENCE

BOTH LEARNING AND REUSE NEED TO BE INTEGRATED INTO AN OVERALL SOFTWARE EVOLUTION MODEL

WHY REUSE HAS BEEN A PROBLEM?

NEED TO REUSE MORE THAN CODE

REUSE OF EXPERIENCE HAS BEEN TOO INFORMAL

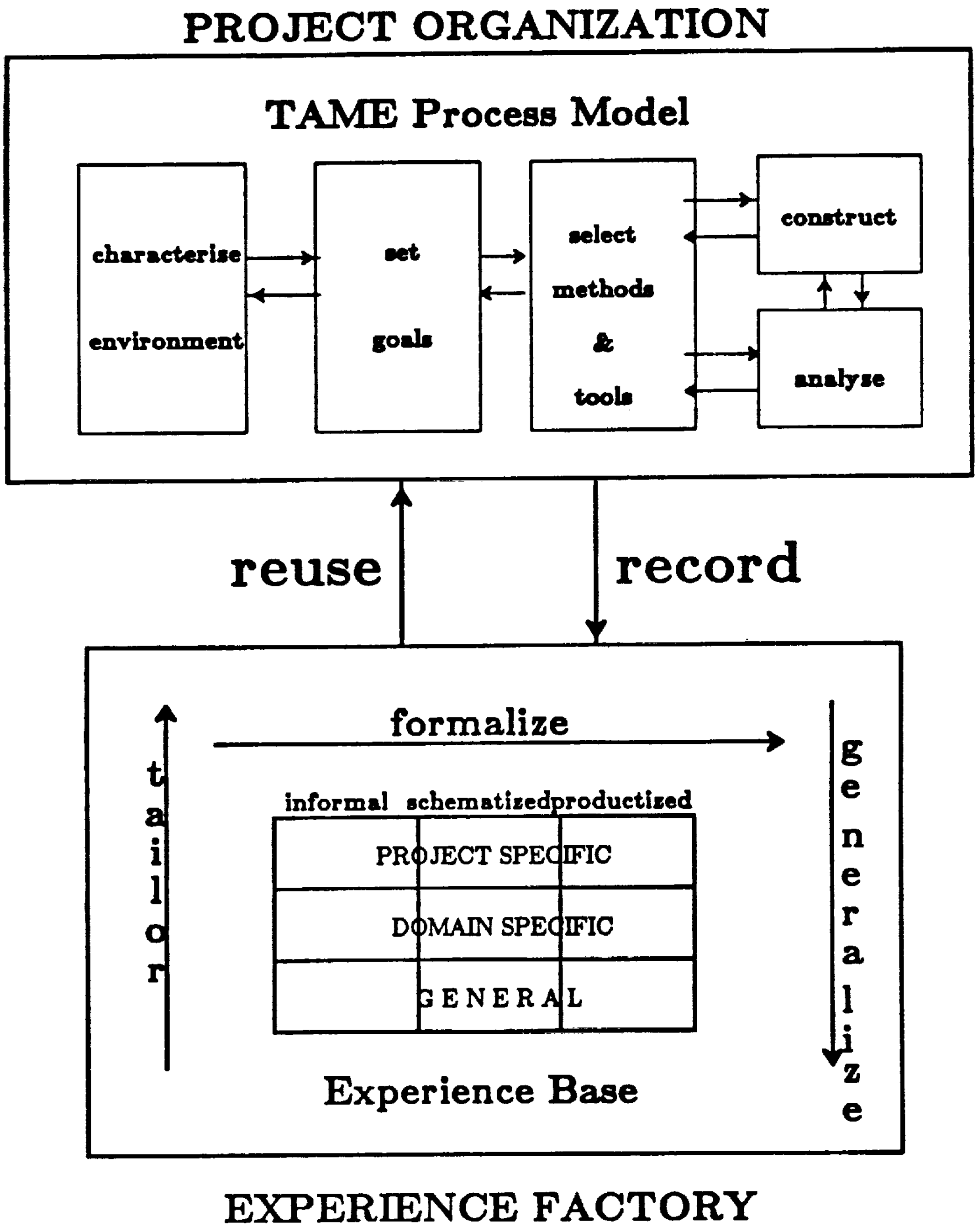
REUSE NOT FULLY INCORPORATED INTO THE PROCESS MODEL

EXPERIENCE NEEDS TO BE TAILORED

EXPERIENCE NEEDS TO BE PACKAGED

EXPERIENCE NEEDS TO BE ANALYZED FOR ITS REUSE POTENTIAL

PROJECT FOCUS IS DELIVERY, NOT REUSE



PROJECT ORGANIZATION

TAME Process Model

characterise environment

set goals

select methods & tools

construct

analyse

reuse

record

formalize

tailor

generalize

informal schematized productized

PROJECT SPECIFIC

DOMAIN SPECIFIC

GENERAL

Experience Base

EXPERIENCE FACTORY

"

EXPERIENCE FACTORY

LOGICAL AND/OR PHYSICAL ORGANIZATION THAT
- SUPPORTS PROJECT DEVELOPMENT BY
  - ANALYZING AND SYNTHESIZING ALL KINDS OF EXPERIENCE
  - ACTING AS A REPOSITORY OF SUCH EXPERIENCE
  - SUPPLYING THAT EXPERIENCE TO VARIOUS PROJECTS ON DEMAND

- PACKAGES EXPERIENCE BY BUILDING
  - INFORMAL, FORMAL OR SCHEMATIZED, AND PRODUCTIZED
    - MODELS AND MEASURES
  - OF VARIOUS SOFTWARE PROCESSES, PRODUCTS, AND
    - OTHER FORMS OF KNOWLEDGE
  - VIA PEOPLE, DOCUMENTS, AND AUTOMATED SUPPORT

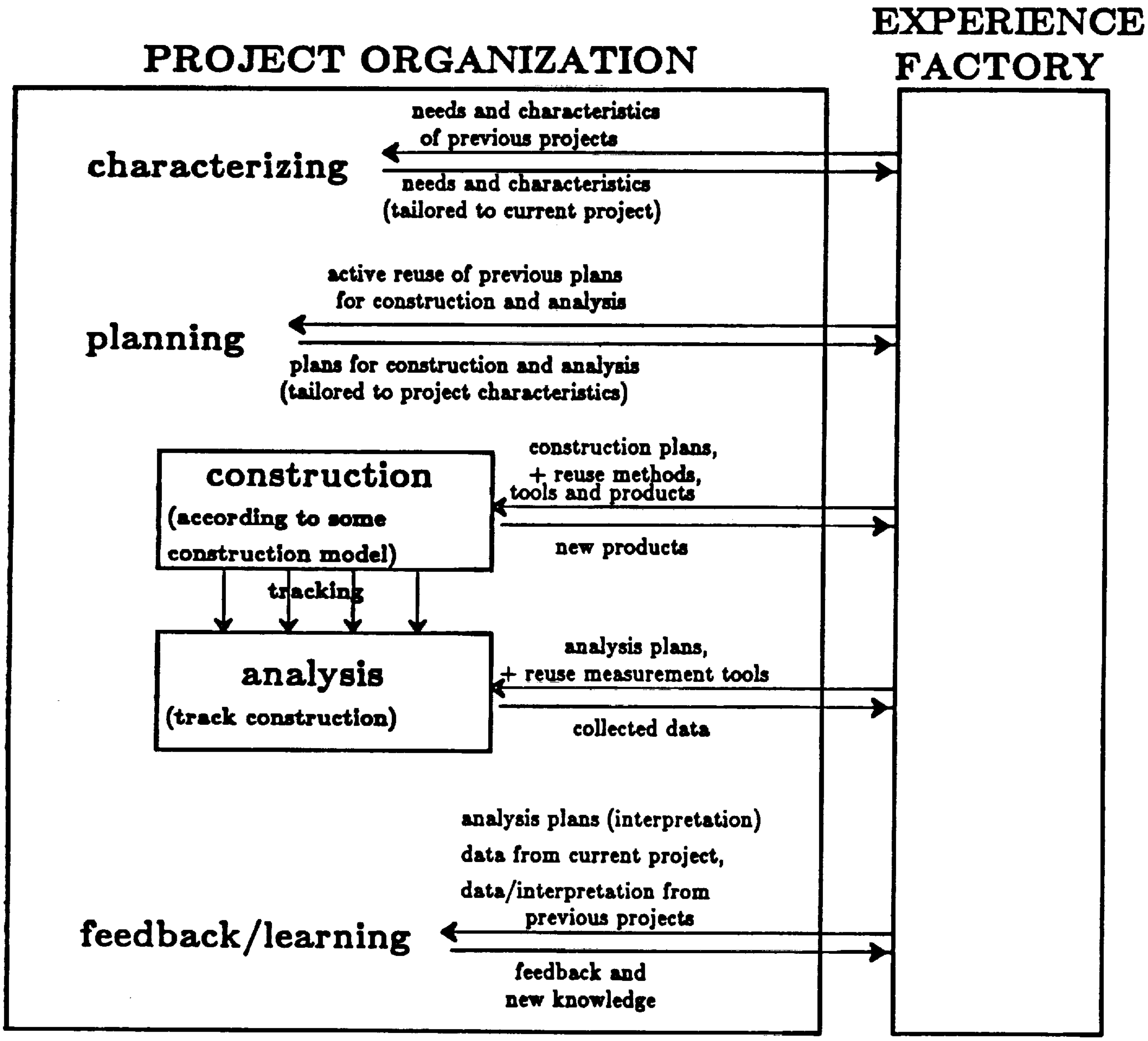
EXPERIENCE FACTORY

SEPARATE ORGANIZATIONS

- PROJECT ORGANIZATION
- EXPERIENCE FACTORY

WHY?

- DIFFERENT FOCUS/PRIORITIES
- DIFFERENT PROCESS MODELS
- DIFFERENT EXPERTISE REQUIREMENTS



PROJECT ORGANIZATION

EXPERIENCE FACTORY

characterizing

needs and characteristics of previous projects

needs and characteristics (tailored to current project)

planning

active reuse of previous plans for construction and analysis

plans for construction and analysis (tailored to project characteristics)

construction (according to some construction model)

construction plans, + reuse methods, tools and products

new products

tracking

analysis (track construction)

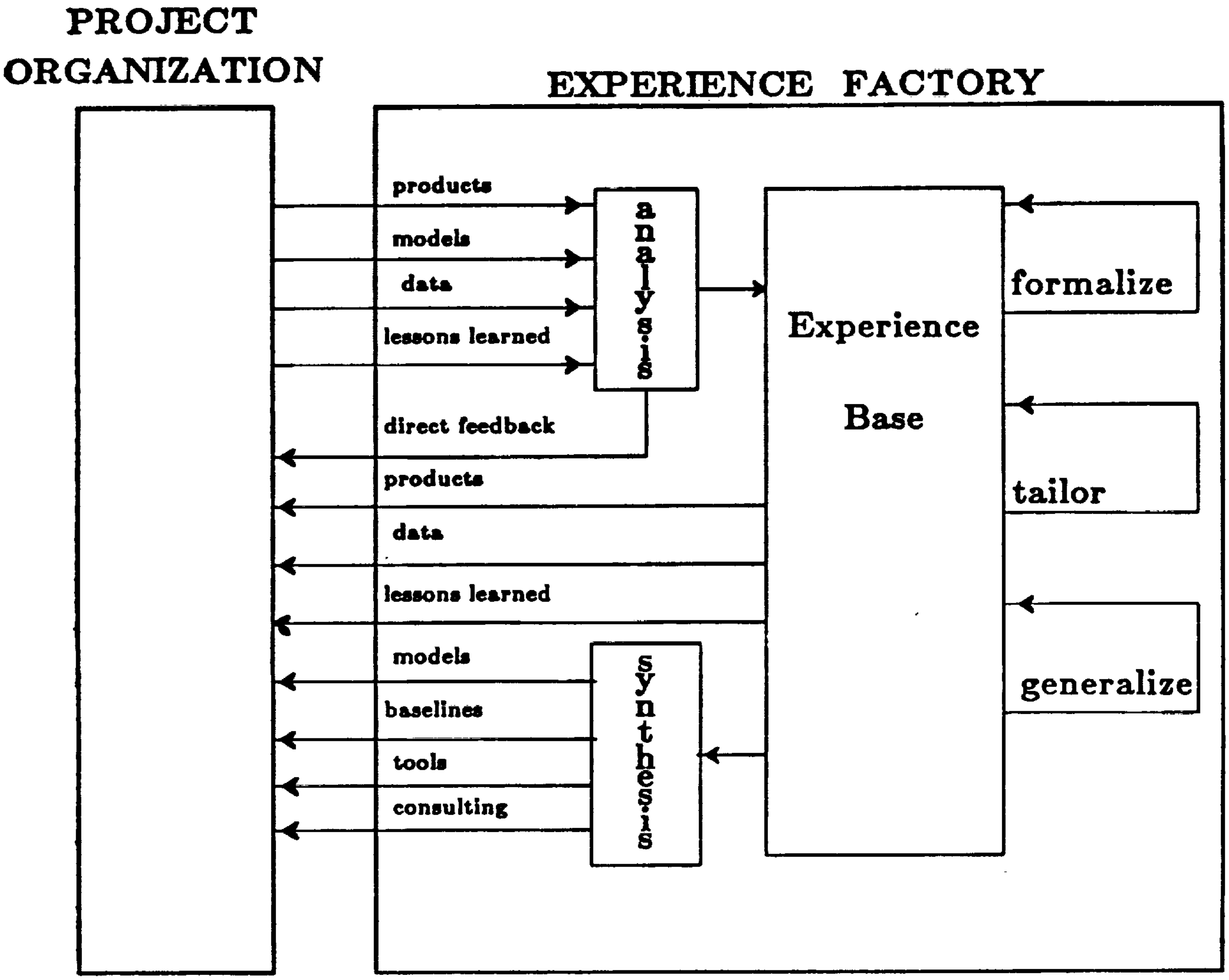
analysis plans, + reuse measurement tools

collected data

feedback/learning

analysis plans (interpretation)
data from current project,
data/interpretation from previous projects

feedback and new knowledge


PROJECT ORGANIZATION
EXPERIENCE FACTORY
products
models
data
lessons learned
analysis
direct feedback
products
data
lessons learned
models
baselines
tools
consulting
synthesis
Experience Base
formalize
tailor
generalize

WHAT KINDS OF EXPERIENCES CAN WE PACKAGE?

PROCESS MODELS

- SEL/STANDARD MODEL FOR GROUND SUPPORT SOFTWARE
- SEL/ADA PROCESS MODEL
- SEL/CLEANROOM PROCESS MODEL

METHOD AND TECHNIQUE DEFINITION/EVALUATION

- READING VS. TESTING
- FUNCTIONAL VS. OBJECT-ORIENTED DESIGN
- ADA VS. FORTRAN

RESOURCE BASELINES/MODELS

- RESOURCE ALLOCATION MODELS
  - STAFFING
  - SCHEDULE
  - COMPUTER UTILIZATION
- COST MODELS AND FACTORS
- RESOURCE/FACTOR RELATIONSHIPS
- TECHNOLOGY/DEFECT ANALYSIS

CHANGE AND DEFECT BASELINES/MODELS
DEFECT BASELINES BY VARIOUS CLASSIFICATIONS
CHANGE BASELINES BY VARIOUS CLASSIFICATIONS
TECHNOLOGY/DEFECT ANALYSES MODELS
DEFECT PREDICTION MODELS

PRODUCT BASELINES/MODELS
GROWTH/CHANGE HISTORIES/ESTIMATION
SIZE/CHARACTERISTIC HISTORIES/ESTIMATION
TEST COVERAGE
REUSE TRADEOFFS

PRODUCTS
APPROPRIATELY "PARAMETERIZED" CODE COMPONENTS
DESIGNS
SPECIFICATIONS
REQUIREMENTS
TEST PLANS

IMPLICATIONS

SEPARATION OF CONCERNS/FOCUS

SUPPORT FOR LEARNING AND REUSE

GENERATES A TANGIBLE CORPORATE ASSET

FORMALIZATION OF MANAGEMENT AND DEVELOPMENT TECHNOLOGIES

CAN START SMALL AND EXPAND

LINKS FOCUSED RESEARCH WITH DEVELOPMENT

IMPLICATIONS

CONSOLIDATION OF ACTIVITIES

PACKAGED EXPERIENCE
CONSULTING
QUALITY ASSURANCE
EDUCATION AND TRAINING

FUNDING ISSUES

SEPARATE COST CENTERS
CORPORATE OVERHEAD
PROJECT BILLED FOR PACKAGES

CONCLUSIONS

COMBINING THE

- IMPROVEMENT PARADIGM
- GOAL/QUESTION/METRIC PARADIGM
- EXPERIENCE FACTORY ORGANIZATION

PROVIDES A FRAMEWORK FOR SOFTWARE ENGINEERING DEVELOPMENT, MAINTENANCE, AND RESEARCH

TAKES ADVANTAGE OF THE EXPERIMENTAL NATURE OF SOFTWARE ENGINEERING

BASED UPON OUR SEL EXPERIENCE

HELPS US

- UNDERSTAND HOW SOFTWARE IS BUILT AND WHERE PROBLEMS ARE
- DEFINE AND FORMALIZE MODELS OF PROCESS AND PRODUCT
- EVALUATE PROCESS AND PRODUCT WITH RESPECT TO SUCCESS CRITERIA
- FEEDBACK TO CURRENT AND FUTURE PROJECTS
- PACKAGE AND REUSE SUCCESSFUL EXPERIENCE

CAN BE APPLIED NOW AND EVOLVE WITH TECHNOLOGY IN A NATURAL WAY

EXPERIENCES IN THE SOFTWARE ENGINEERING LABORATORY (SEL) APPLYING SOFTWARE MEASUREMENT

by Frank McGarry, Sharon Waligora, and Tim McDermott

INTRODUCTION

The Software Engineering Laboratory (SEL) was established in 1977 as a cooperative effort among the National Aeronautics and Space Administration's (NASA's) Goddard Space Flight Center (GSFC), Computer Sciences Corporation (CSC), and the University of Maryland to understand and improve the software development process and its products within GSFC's Flight Dynamics Division. During the past 14 years, the SEL has collected and archived data on over 100 software development projects in the organization. This has allowed the SEL to gain an understanding and to model the development process. From these data, the SEL has derived models and metrics that describe the typical flight dynamics software development process. These models and metrics are the basis for software estimation, planning, and general management in this environment. They also provide typical project profiles against which ongoing projects can be compared and evaluated. The SEL provides managers in this environment with tools (on-line and paper) for monitoring and assessing project status.

This paper presents experiences in the SEL of applying software measurement. Examples from flight dynamics project data are presented that demonstrate how the SEL has used software measures to (1) understand the local software environment, (2) manage active production projects, (3) plan future projects, and (4) develop rationale for adopting software standards and technology.

SEL Product Environment (Viewgraph 2)

The SEL production environment consists of projects that are classified as mid-sized software systems. The average project lasts 26 months and requires 9.5 staff years of effort. The average project develops 93,000 source lines of code (SLOC) and delivers 102,000 SLOC.

Virtually all projects in this environment are scientific ground-based systems although some embedded systems have been developed in this environment. The bulk of the software is developed in FORTRAN although Ada is starting to be used more heavily, while other languages, such as Pascal and assembly, are used occasionally.

The average staff level for a typical SEL project is 5.4 full-time people. SEL managers average 10 years of overall experiences, with 5.8 years in the application area, and the technical staff averages 8.5 years overall experience, with 4.0 years in the application.

Software Technology Studies in the SEL (Viewgraph 3)

The SEL has undertaken many technology investigations since 1977. Data have been collected on more than 75 production software development projects, and all of these data have been fed back into the SEL's experience base.

The SEL regularly collects detailed data from all its development projects. The types of data collected include cost (measured in effort), process data, and product data. The process data include information about the project, such as the methodology, tools, and techniques used, and information about personnel experience and training. Product data include size (in SLOC), change and error information, and the results of postdevelopment static analysis of the delivered code. For a more detailed description of the data collected, see *Data Collection Procedures for the Rehosted SEL Database*, SEL-87-008.

The SEL has analyzed over 50 technologies, such as design approaches, testing techniques, tools, environments, training, languages, and methodologies. Also, the SEL has published more than 150 papers and reports detailing the results of these investigations.

In the feedback process, the SEL has evolved the standards and practices used for Flight Dynamics software development. These include models of effort, changes and errors, and costing. The SEL has also established quality assurance procedures and testing strategies. Standards and practices are an important avenue of feedback of the measurement performed by the SEL.

SEL APPLICATION OF MEASUREMENT

There are four major applications of measurements within the SEL:

Understanding the Software Environment is essential to any software engineering undertaking. Before anything can be changed, it must be understood the way it exists now.

Management of Current, Active Projects depends on measuring the projects and on having a baseline of experience against which to compare projects trends and absolute measures.

Planning Future Projects requires cost models, standard effort distributions, assessments of available technologies, and scheduling models, all of which depend on measurement and a clear understanding of the environment.

Rationale for Adopting Software Standards and Technology is the least obvious, but arguably the most important, application of measurement. Measurement allows the SEL to quantitatively evaluate new technologies that have potential for favorably affecting the SEL environment. Through this method, appropriate technology can be inserted quickly and large-scale misapplication of inappropriate techniques (for the SEL) can be avoided.

The remainder of this section discusses each of these areas in more detail.

UNDERSTANDING THE LOCAL SOFTWARE DEVELOPMENT ENVIRONMENT

Understanding what an organization does and how the organization operates is fundamental to any attempt to plan, manage, or improve the organization. This is true in general and especially true for software development organizations. The following examples illustrate how the SEL has come to understand its environment. The measures examined are certainly not exhaustive but show how understanding comes from measurement.

Where Do Developers Spend Their Time (Viewgraph 6)

There are two majors points to this chart. The first point is that the baseline characteristics of the development process must be understood if projects are to be planned and managed or if new technology is to be evaluated. The second point is that a stable environment is not quickly or easily upgraded by changes to the process.

One baseline characteristic of the SEL software development process is effort distribution, that is, which phases of the life cycle consume what portion of development effort. Viewgraph 6 compares the distributions of effort for FORTRAN and Ada projects in the SEL, both by life-cycle phase and by activity. The phase data counts hours charged to a project during each phase. The activity data counts all hours attributed to a particular activity, regardless of when in the life cycle the activity occurred. Understanding these distributions is important to assessing the progress of an ongoing project, planning new efforts, and even evaluating new technology. The Ada distributions are a case in point.

These graphs of effort by activity show that, contrary to the early expectations for Ada, there has been no radical change in programmers' effort distribution. Ada projects spend about 20 percent of their effort on design, versus a slightly higher figure (23 percent) for FORTRAN. The comparison for coding is 18 percent versus 21 percent, for testing it is 34 percent versus 30 percent. "Other," the final category, is 27 percent of Ada effort versus 26 percent of FORTRAN. "Other" includes all of the ancillary activities that do not fit into one of the primary categories, such as managing, training, attending meetings, and documenting.

The graphs of effort by phase shows some change in the Ada distribution. The design phase takes 27 percent of the Ada effort, versus 26 percent for FORTRAN.

Ada code phase consumes 46 percent of total effort, compared to only 37 percent for FORTRAN. The test phase takes 27 percent in Ada and 37 percent in FORTRAN. SEL experience indicates that, in this environment at least, there is a legacy of many years of developing FORTRAN systems that is not quickly changed, not even by such a significant change to the process as using a different language like Ada.

Comparative Classes of Errors (Viewgraph 7)

Comparison of the types of errors that are being made in FORTRAN and Ada projects gives similar results. Again, contrary to expectations, there seems to be little difference in the error profiles observed in systems using the two languages. Computational and initialization errors are each 15 percent of the errors for both languages. Data errors differ by only 1 percent, 31 percent for Ada as opposed to 30 percent for FORTRAN. Logic or control errors are higher in Ada, 22 percent versus 16 percent, while interface errors are lower, 17 percent for Ada versus 24 percent for FORTRAN.

The SEL is learning through measurement that the long heritage of FORTRAN development is not easily changed. The way the organization does business and the experiences of the individuals in the organization is a stronger influence on the performance of a project than any one specific technology.

Software Growth Profile in the SEL (Viewgraph 8)

The software growth profile in the SEL is a good example of the models that are developed to understand the local environment. Lines of code are not counted in this growth model until they are placed in controlled libraries.

Typically, only a small amount of code is developed during the design phase and the first part of implementation. SLOC growth during implementation shows periods of sharp growth separated by more moderate growth. This is a reflection of the SEL practice of implementing systems in builds. Also, in this environment, developers tend to retain code until they can deliver integrated chunks of the system to the controlled libraries, a practice which contributes to the surges in code growth.

This model also shows that, typically, 10 percent of the code is produced after the start of testing. Measuring code growth led the SEL to investigate why the system continues to grow after the end of implementation. The growth reflects error corrections and enhancements made to make the system more suitable to the needs of the users. Measurement focused attention on this growth and led to a deeper understanding of the way the SEL does business.

Error Detection Rate in the SEL (Viewgraph 9)

The error detection rate is another interesting model from the SEL environment. There are two types of information in this model. The first is the absolute error rates expected in each phase. The rates shown here are based on projects from the mid-1980s. The SEL expects about four errors per thousand SLOC during implementation, two during system test, one during acceptance test, and one-half during operation and maintenance. Analysis of more recent projects indicates that error rates are declining as the software development process and technology improve.

The second piece of information is that the error detection rates reduce by 50 percent in each subsequent phase. This datum seems to be independent of the actual values of the error rates. It is still true in the recent projects where the overall error rates are declining. The next section will show how this understanding can be applied.

MANAGEMENT OF ACTIVE DEVELOPMENT PROJECTS

Once an environment is understood, historical data can be used to develop models that describe the expected behavior of the "typical" project. Managers in the SEL compare current trends of active project data with expected trends (models) and those of similar past projects to assess the current state of their project. Effort, computer utilization, error and change rates, and size estimates are among those data that SEL managers find most useful in assessing stability, quality, and reliability. The following paragraphs illustrate management through measurement in the SEL.

Using Software Error Rates (Viewgraph 11)

This example shows the use of the error rate model on the Cosmic Background Explorer (COBE) attitude ground support system (AGSS). Comparing the measured error rate with the SEL model described in Viewgraph 9 gives an early indication of the quality of the product. In this case, both COBE's absolute error rates and the decline in the detection rate are better than the model, an occurrence which gives a strong indication that this system will be more reliable than average. In fact, the software for this project has proven to be extremely reliable.

If the error rate had been low, but the detection rate had not declined, SEL experience would have pointed to inadequate testing and a less reliable system.

Tracking System Failure Reports (Viewgraph 12)

This graph shows how failure reports behaved during acceptance testing of one project. Early in acceptance testing most of the staff effort is spent performing and evaluating tests. If the system is as reliable as planned, the failure rate will decline as testing proceeds, allowing the staff to spend more effort fixing defects. In this case, twice the expected numbers of errors were found during acceptance testing. Fifteen weeks into testing nearly all of the expected errors had been detected and hardly any had been fixed; clearly not enough staff were allocated to fixing problems. In the 17th week, additional staff was allocated to correct errors.

Almost immediately, the open failure reports ("X" curve) flattened out and began to decline as the fix rate accelerated and the error detection rate slowed down. The point at which the "open" and "fixed" curves cross is especially important because it marks the point at which defects are being repaired faster than they are being discovered. At this point a manager can more confidently predict the end of acceptance testing.

Tracking Computer Use (Viewgraph 13)

This example compares a typical SEL project's use of central processing unit (CPU) resources on the left side of the chart to a project with a deviant CPU use profile on the right. Being different does not mean that the project is necessarily in trouble. For example, the project might be using the cleanroom methodology; the project might be doing extensive desk work. Here, the CPU usage curve told the project's managers that something was different and raised a flag that this project should be examined. In this case, investigation showed that the project was being adversely affected by a high number of to be determined (TBD) requirements, requirements changes, and redesign. Management replanned the project, taking these factors into account.

CPU usage data are an example of valuable data that are easy to collect. Most operating systems have accounting systems that provide it. However, for CPU usage data or any other measurement to be useful to the management of development projects, a baseline model must explain the behavior of the measure in the local environment.

Characteristic Staffing Profiles (Viewgraph 14)

This is the time distribution of effort on two projects of similar complexity. The profile on the left is typical for the SEL, with peaks near the beginning and end of the implementation phase.

The project on the right suffered from the Mythical Man Month syndrome. Responding to significant project requirements changes in the middle of implementation, the staff was nearly tripled to try to meet schedule requirements. Staff levels did not start to decline until the start of acceptance testing. Both productivity and reliability suffered on this project.

A staffing profile with a sharp increase late in the development life cycle is a clear indicator that something on the project is out of control and that quality and reliability will likely be lower than expected.

Tracking Estimates of Size at Completion (Viewgraph 15)

Tracking final size estimates provides another strong management indicator. Project 1, on the left, had a typical SEL history of manager's estimates of the final size of the system. In the SEL environment, requirements changes and specification modifications usually cause a system to grow up to 40 percent larger than the estimates made at preliminary design review (PDR).

Project 2, on the right, shows several deviations from the normal trend. It experienced extreme inflation of the size estimates in the middle of the code phase. The spot labeled 2 on the graph represents an increase of nearly 25 percent in the manager's estimate of the final size. This should have caused a management review of the project. No action was taken, and the underlying causes of the inflation, primarily specification changes, continued to increase the size of the project. Finally, at the spot labeled 3 on the graph, following another 50 percent increase of the size estimate, the project underwent a detailed management review, and the changes were brought under control.

Management should have questioned the decrease in size estimates at the critical design review (CDR) (label 1 on the graph) after the size had grown significantly during preliminary design. This was an early indicator that the specifications were not as stable as is expected in this environment.

Using Effort Data in Replanning (Viewgraph 16)

Effort data can be a significant aid in replanning, as illustrated by the history of successive staffing plans for one project.

The SEL has discovered two typical effort distributions for this environment. One of them is roughly parabolic and the other has two peaks: the first near CDR and the other near the start of testing.

The first schedule was based on an underestimate of the size of the system and used a rough parabola for effort distribution. Toward the end of design, it became clear that the system was larger than anticipated, and the effort was replanned at CDR. The first replan used the SEL two-peak model of effort distribution. Effort continued to grow when the second plan called for it to level off and decline. An audit was held in the middle of the code phase when it was clear that still more staff were required to maintain progress. The audit determined that the project was plagued with an unusually large number of unresolved TBDs and requirements changes and that—as part of the corrective action—another replan was necessary. The second replan was based on an accurate size estimate and returned to the parabolic distribution, which the project followed to a successful completion.

This is a straightforward example of the use of metrics data in both planning and monitoring a project. The relationships that have been documented for this environment support planning and the collection of data on the performance of current projects allows corrective action to be taken before projects are hopelessly off target.

FOUNDATION FOR PLANNING FUTURE PROJECTS

A vital application of measurement is planning future projects. The models and relationships that emerge from measuring the local environment are the basis of sound estimates and plans.

Planning Aids for One Environment (Viewgraph 18)

The cost estimation equation, the effort distribution model, the computer utilization estimation equation, and the documentation estimation equation are examples of the relationships that are used as planning aids in the SEL environment. They allow managers to generate realistic project plans, with proper allocation of effort and scheduling of milestones. Supporting resources can be planned and scheduled with confidence.

Equations that produce an estimate of the cost of a project from an estimate of the size of the system are widespread today. The SEL version of this equation is $\text{Effort} = 1.48 * \text{KSLOC}^{98}$. An estimate of total effort is not enough, however. Effective planning requires an understanding of how effort will be spent and when reviews and milestones should occur. The effort distribution model provides SEL managers with schedule guidance. Other relations are also important to good planning, such as how much computer resources will be required and how much documentation will be published. SEL project data are evaluated periodically to produce up-to-date planning data for new projects.

Additional Local Planning Aids (Viewgraph 19)

Software managers in the SEL have observed relations—such as the fraction of changes that are due to errors, the growth of size estimates over the life of projects, and the cost of maintenance—that are useful for planning. Approximately one-third of the changes made in this environment are made to correct errors. This heuristic is useful for gauging the quality of the system as it is developed and assessing the effectiveness of testing. The final system is about 40 percent larger than the size estimate at PDR. This observation quantifies the stability of the requirements of the systems built in the flight dynamics area. Maintenance costs per year are about 12 percent of the original development cost. These rules grow out of understanding the environment.

This chart also presents metric relations between FORTRAN and Ada. Ada programs that conform to the SEL Ada style guide have three times the SLOC of their FORTRAN equivalents. When comments are discounted, Ada programs are 2.5 times the size of their FORTRAN equivalents. When only those source lines that are part of an executable statement are counted, Ada programs are still twice as large as FORTRAN. Finally, when considering only the number of statements, the programs in the two languages are the same size.

These differences between Ada and FORTRAN reflect the coding styles used for Ada and FORTRAN in the SEL. One of the important lessons to be learned from this viewgraph is that even in a baselined environment, it is crucial to understand which model to use for planning.

RATIONALE FOR ADOPTING PRACTICES AND GUIDELINES

Measurement provides an organization with justification for the way it does business and an orderly process for selecting which new technologies and methods to adopt. This section presents the SEL experience with evaluating and trying to understand two sample technologies that are candidates for becoming "standard" in the SEL. These two technologies, the Ada language and Object-Oriented Development (OOD), are currently under intense study in the SEL.

Impacts of Ada on a Production Environment (Viewgraph 21)

This chart shows the current results of the SEL's investigation of Ada. The adoption of Ada is much more than just changing languages. Proper use of Ada implies the use of new software engineering techniques that must be learned and practiced. This deeper change adversely affected the productivity results from early Ada projects. While initial productivity results for Ada were below baseline FORTRAN productivity, the trends are in the right direction in subsequent uses of the technology.

The use of Ada has demonstrated sufficient positive residual effects to offset initial productivity concerns. The use of Ada seems to have favorably affected some measures in this environment; for example, Ada technology has improved the level of reuse for the sample set of projects studied so far.

The history of the insertion of Ada technology into the SEL environment shows that in some cases the environment must evolve to be able to effectively utilize this new technology. The FORTRAN legacy in the SEL, such as life-cycle models, estimating relations, and review techniques, is pervasive and changes slowly.

Object-Oriented Development (OOD) and Code Reuse (Viewgraph 22)

This viewgraph contrasts the levels of reuse achieved by five recent projects using structured analysis (SA) and five projects using OOD. The SA projects seem to stay relatively constant in the level of about 35 percent reuse, even when reuse is pursued very aggressively (Upper Atmosphere Research Satellite (UARS) Dynamics Simulator (UARSDS) project). The OOD projects, however, start with better than 30 percent reuse. With experience, and the accumulation of a body of Ada code for reuse, the last two OOD projects are projecting 76 percent and 90 percent reuse. The significant level of reuse in the Extreme Ultraviolet Explorer (EUVE) Telemetry Simulator (EUVETELS) (90 percent) was accomplished through reusing specifications and design as well as code. OOD seems well suited for reuse, but further study is required to conclude that OOD technology is primarily responsible for these high levels of reuse.

OOD, in the SEL, is a new technology success story. The SEL has tried and abandoned many other technologies, but currently intends to keep working with OOD. The keys to being able to evaluate new approaches to software development are (1) being able to measure trial projects and (2) having a baseline of the environment against which to judge the new method.

CLOSING THE LOOP—APPLYING MEASURED TECHNOLOGIES IN THE SEL

Viewgraph 23 shows examples of how the SEL closes the feedback loop by incorporating the results of studies in the SEL guidelines for software development. This feedback is the primary mechanism of improving the software development process in the SEL. Without feedback, it is not possible to ensure that management capitalizes on the lessons learned on prior projects.

The seven documents listed on the right side of the viewgraph define the set of standards and guidelines currently used as a result of the SEL studies. They contain the SEL life-cycle model, the process model, and the product models. These documents obviously change as the SEL gains more experience.

SUMMARY

This paper has presented some examples of the way that the SEL measures software development and uses the measurements. Measurement produces understanding of the environment. This understanding can then be used in planning and managing projects. Finally, understanding is necessary as a basis for evaluating new tools and techniques so that a continually improving process may be adopted for the software development organization.

VIEWGRAPH MATERIALS

FOR THE

F. MCGARRY PRESENTATION

EXPERIENCES IN THE SEL
APPLYING SOFTWARE MEASUREMENT

FRANK MCGARRY

NASA/GSFC

AND

SHARON WALIGORA
TIM McDERMOTT

COMPUTER SCIENCE
CORPORATION

14TH ANNUAL SOFTWARE ENGINEERING WORKSHOP - NOVEMBER 29, 1989

J217.004

VIEWGRAPH 1

SOFTWARE ENGINEERING LABORATORY PRODUCTION ENVIRONMENT

TYPES OF SOFTWARE: SCIENTIFIC, GROUND-BASED, INTERACTIVE GRAPHIC, MODERATE RELIABILITY AND RESPONSE REQUIREMENTS

LANGUAGES: 75% FORTRAN, 15% Ada, 10% OTHER (PASCAL, C, ALC,...)

PROJECT CHARACTERISTICS:	**AVERAGE**	**(RANGE)***
DURATION (MONTHS)	26.0	(18-43)
EFFORT (STAFF-YEARS)	9.5	(02-30)
SIZE (1000 LOC)		
DEVELOPED	93.0	(25-250)
DELIVERED	102.0	(03-30)
STAFF (FULL-TIME EQUIV.)		
AVERAGE	5.4	
PEAK	10.0	
INDIVIDUALS	24.0	
APPLICATION EXPERIENCE (YEARS)		
MANAGERS	5.8	
TECHNICAL STAFF	4.0	
OVERALL EXPERIENCE (YEARS)		
MANAGERS	10.0	
TECHNICAL STAFF	8.5	

* SAMPLE OF 10 FORTRAN PROJECTS

J217.005

VIEWGRAPH 2

SOFTWARE TECHNOLOGY STUDIES
IN THE SEL
(1977 - 1989)

- 75+ PRODUCTION SOFTWARE PROJECTS STUDIED (EXPERIMENTS)

- DETAILED DATA/MEASURES FROM ALL PROJECTS (DURING DEVELOPMENT)
    - EFFORT (COST)
    - ERRORS
    - CHANGES
    - METHODS
    - PROJECT CHARACTERISTICS
    - TOOLS
    - PERSONNEL
    - TRAINING
    - PRODUCT DATA

- OVER 50 SPECIFIC SOFTWARE TECHNOLOGIES ANALYZED (TESTING, DESIGN, TOOLS, ENVIRONMENT, TRAINING, METHODOLOGIES,...)

- OVER 150 PAPERS/REPORTS (DETAILED ANALYSIS PRODUCED FROM SEL)

- EVOLVING STANDARDS AND PRACTICES (NASA DEVELOPMENT ORGANIZATIONS)
    - EFFORT (COST)
    - ERRORS
    - CHANGES
    - COSTING
    - QUALITY ASSURANCE
    - TESTING STRATEGIES

J217.003

VIEWGRAPH 3

SEL APPLICATION OF MEASUREMENT

1. • UNDERSTANDING THE SOFTWARE ENVIRONMENT
2. • MANAGING ACTIVE PRODUCTION PROJECTS
3. • PLANNING FUTURE PROJECTS
4. • RATIONALE FOR ADOPTING S/W STANDARDS/TECHNOLOGY

J217.007

VIEWGRAPH 4

SEL APPLICATION OF MEASUREMENT

①

UNDERSTANDING THE “LOCAL”

SOFTWARE DEVELOPMENT ENVIRONMENT

J217.008

VIEWGRAPH 5

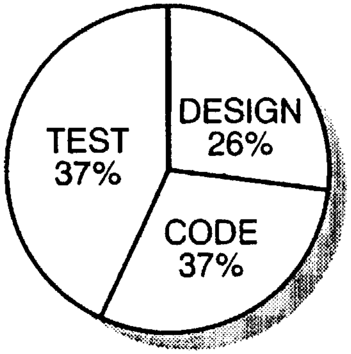
WHERE DO DEVELOPERS SPEND THEIR TIME

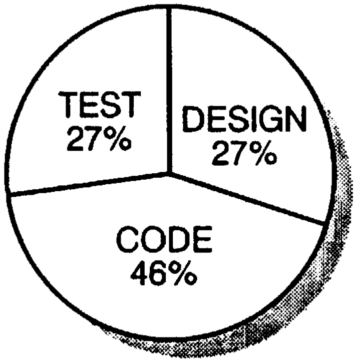
(FOR SIMILAR CLASSES OF SOFTWARE)*

FORTRAN

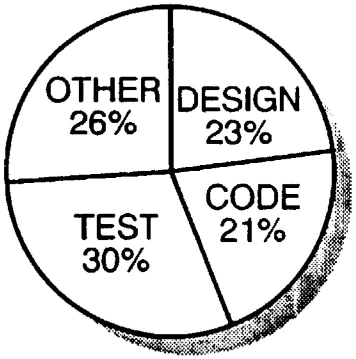
Ada

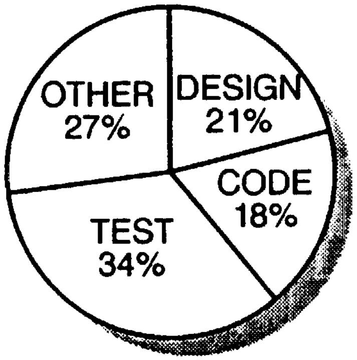
BY LIFE CYCLE PHASE (DATE DEPENDENT)





BY ACTIVITY (NOT DATE DEPENDENT)





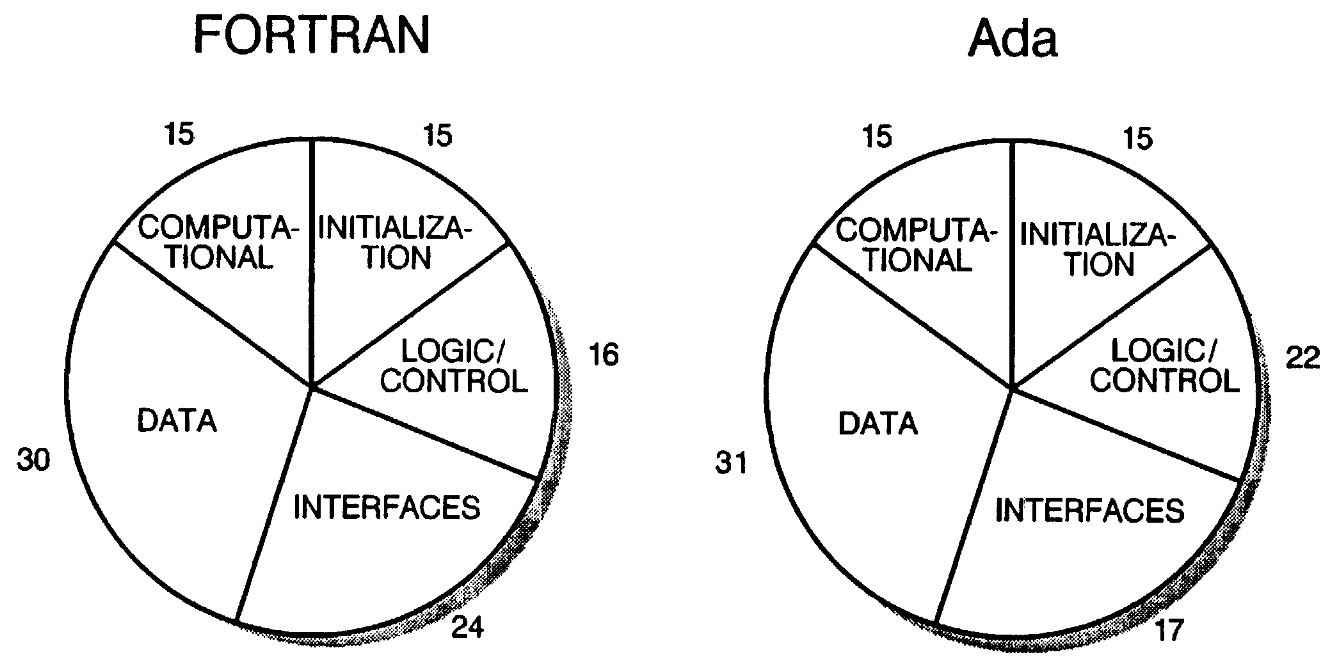
"HERITAGE" OF ENVIRONMENT IS NOT QUICKLY CHANGED

*BASED ON 5 Ada AND 8 FORTRAN PROJECTS

J217.009

VIEWGRAPH 6

COMPARATIVE CLASSES OF ERRORS*


FORTRAN

Ada

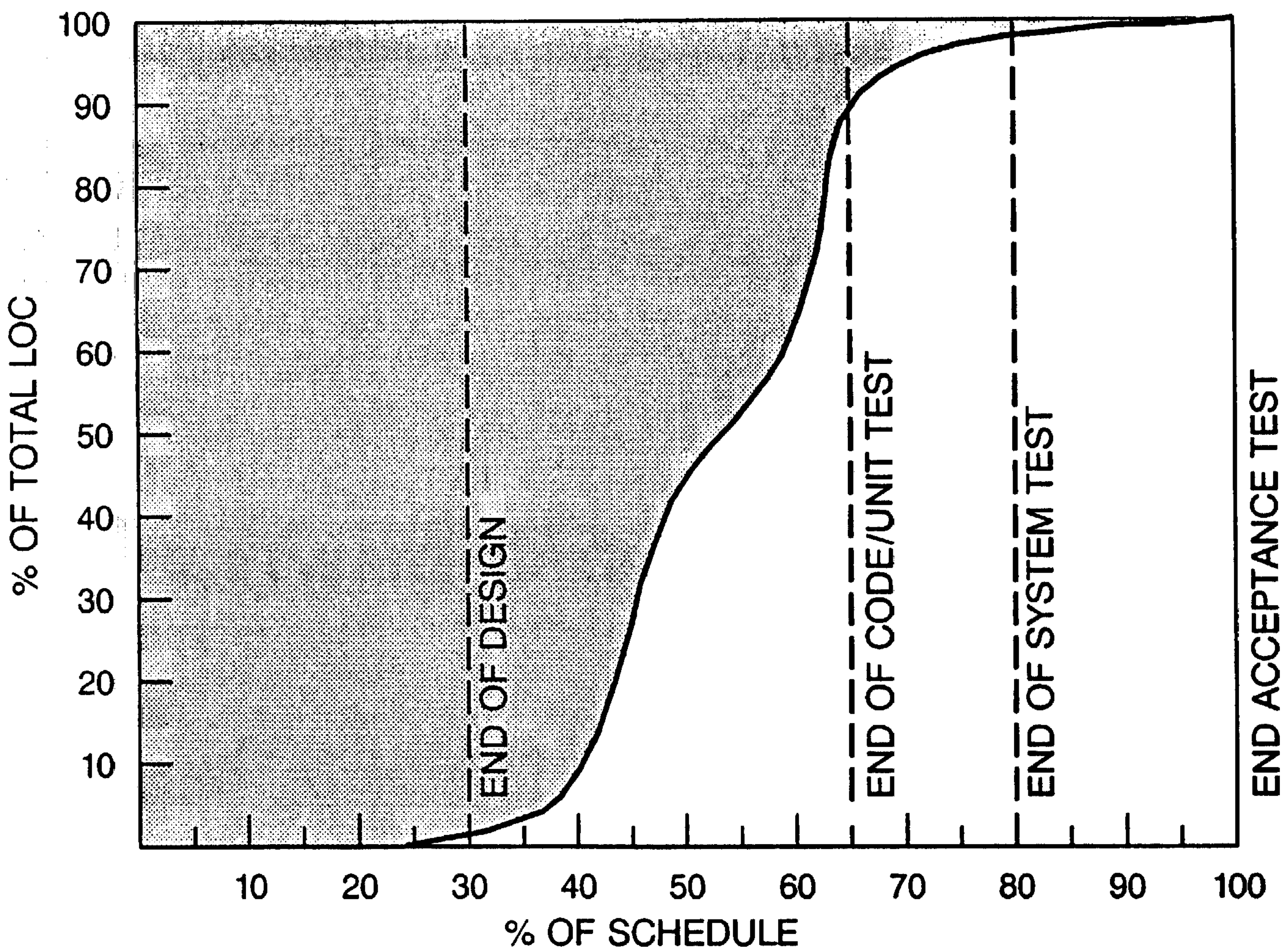
*ERROR PROFILES QUITE SIMILAR; EVEN FOR DIFFERENT LANGUAGES
*Ada SOMEWHAT FEWER INTERFACE ERRORS

*BASED ON ERRORS FROM 5 Ada PROJECTS AND 8 FORTRAN PROJECTS

J217.010

VIEWGRAPH 7

SOFTWARE GROWTH PROFILE IN SEL



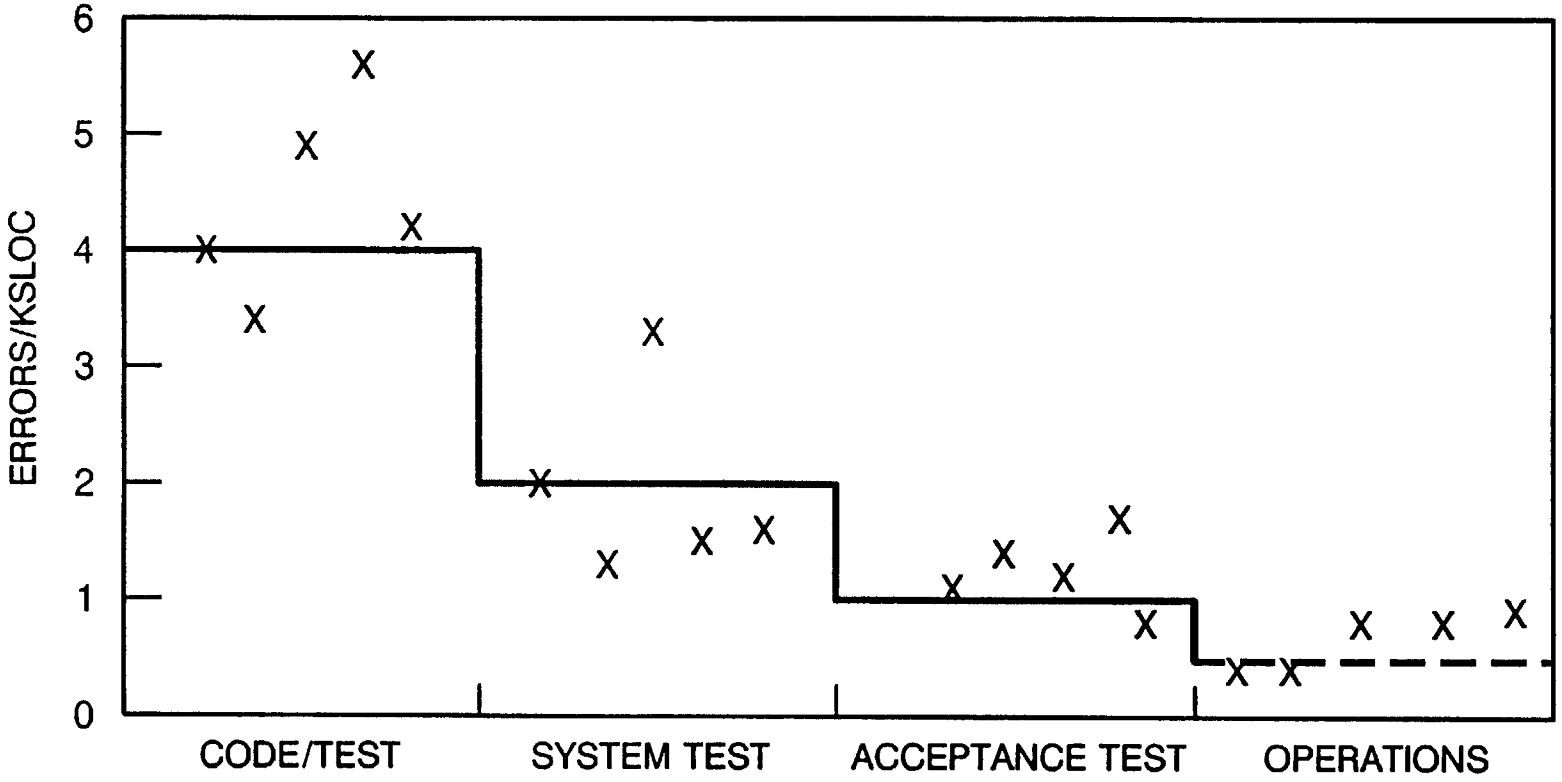
CONSISTENT DEVELOPMENT PATTERNS REFLECT THE CHARACTERISTICS OF THE LOCAL ENVIRONMENT

VIEWGRAPH 8

J217.012

ERROR DETECTION RATE*

(IN ONE ENVIRONMENT)



SEL ENVIRONMENT EXPECTS TO HALVE ERROR RATE IN EACH SUBSEQUENT PHASE

*BASED ON 5 PROJECTS BETWEEN 1983 AND 1987

J217.013

VIEWGRAPH 9

SEL APPLICATION OF MEASUREMENT

2

SUPPORTS THE MANAGEMENT OF

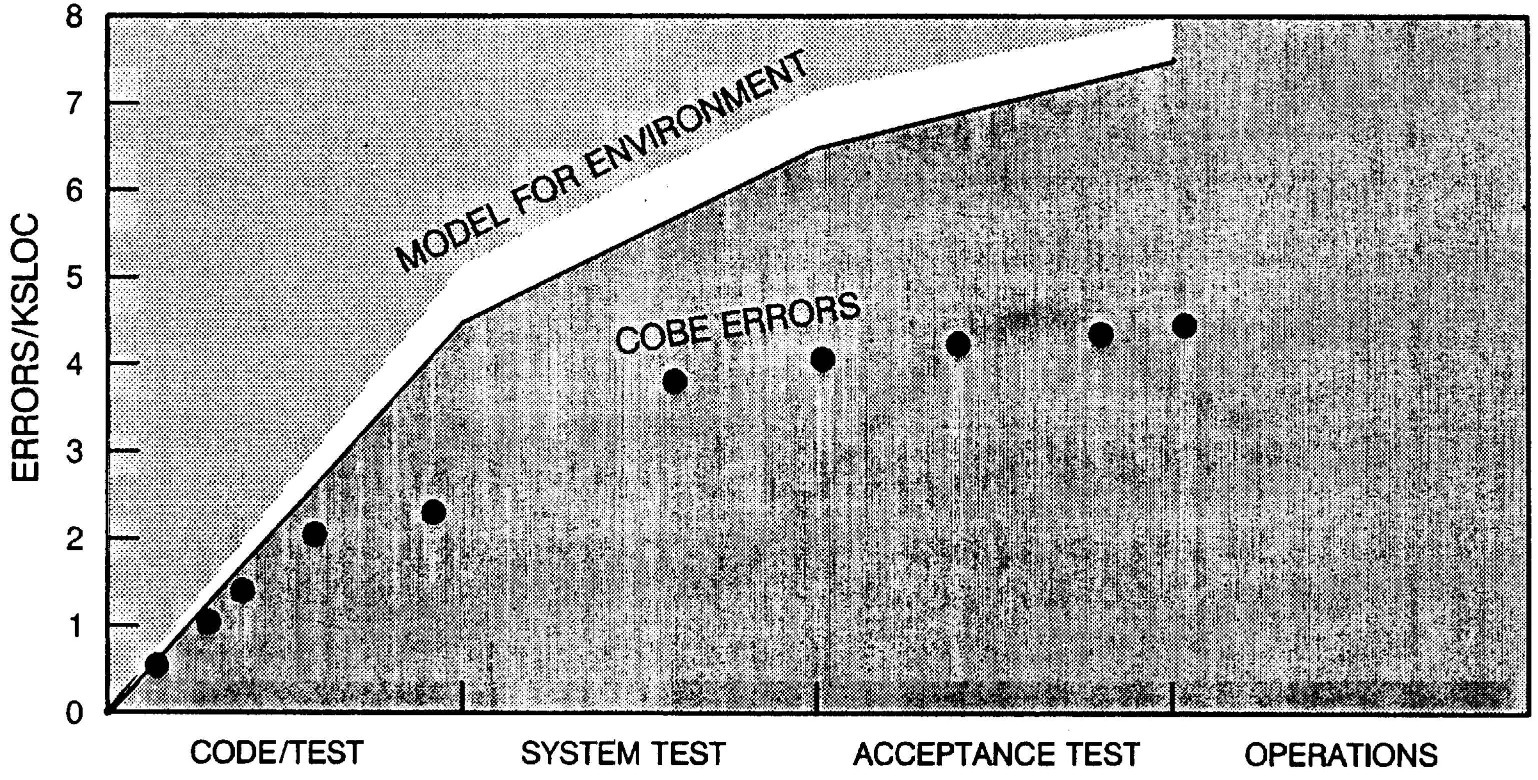
ACTIVE DEVELOPMENT PROJECTS

J217.014

VIEWGRAPH 10

SOFTWARE ERROR RATES

TRACKING "COBE" RELIABILITY



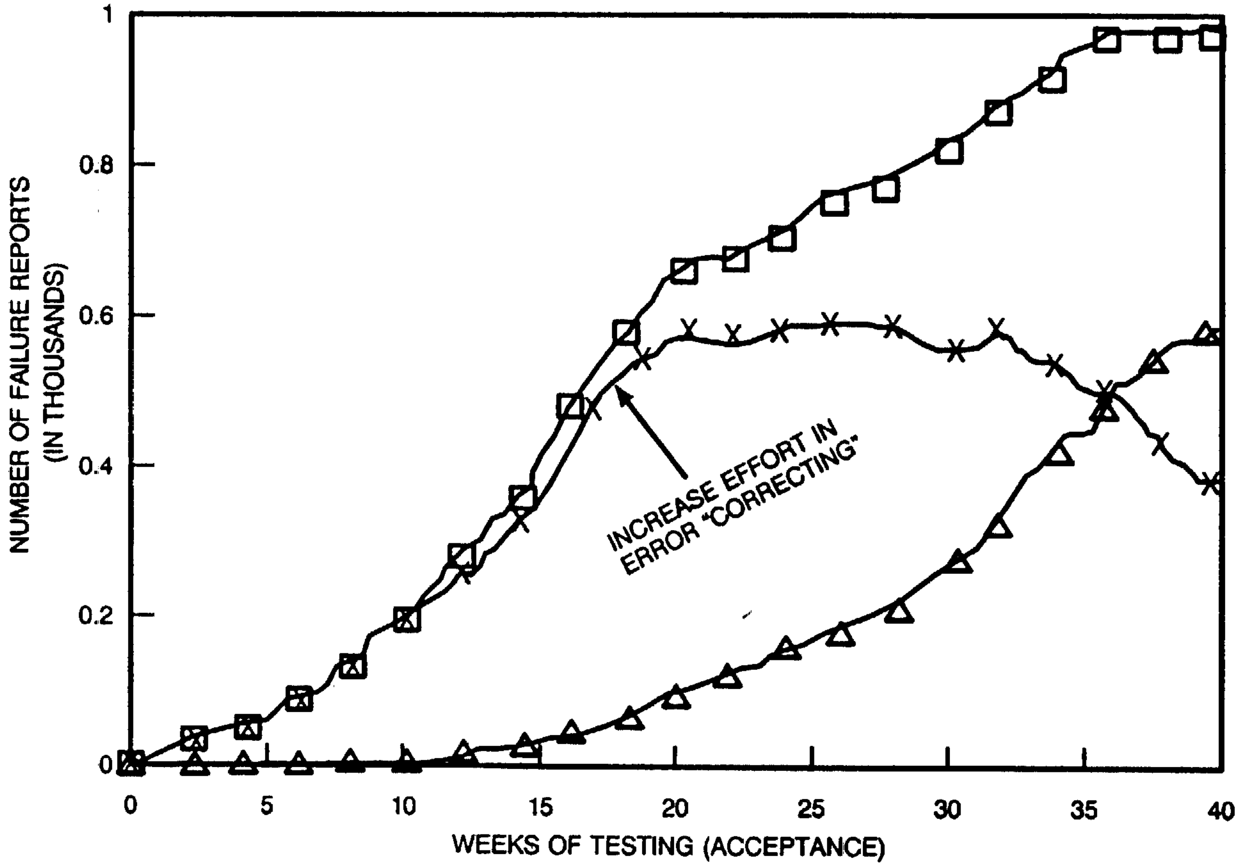
MEASURING ERROR RATES CAN PROVIDE EARLY INDICATION OF SOFTWARE QUALITY

J217.015

VIEWGRAPH 11

SYSTEM FAILURE REPORTS

(FROM 1 PROJECT IN SEL)*



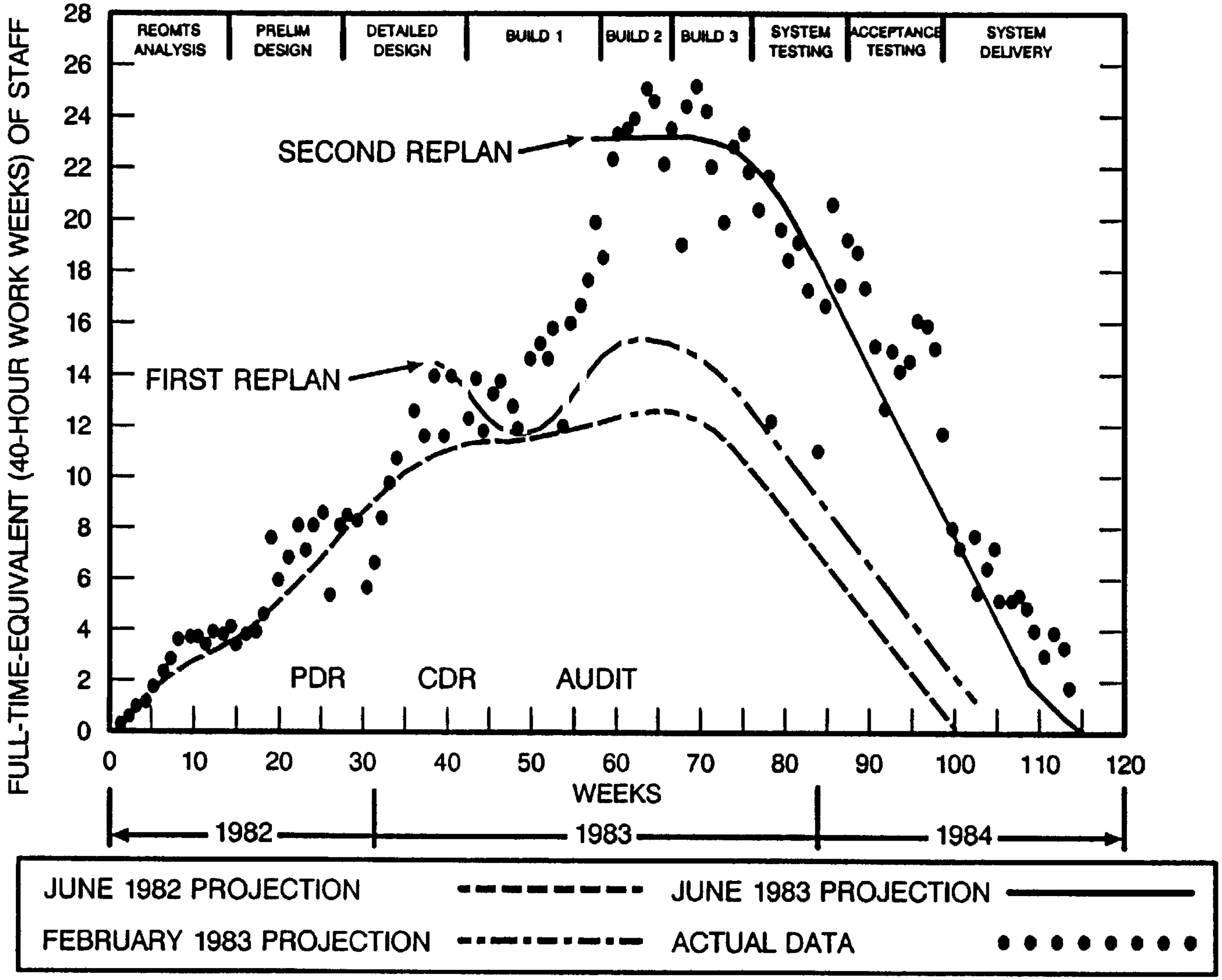
MEASURING FAILURES AND FIX RATES CAN BE A VALUABLE MANAGEMENT TOOL

* "TCOPS" SIZE = 1.2 MSLOC

J217.016

VIEWGRAPH 12

EFFORT DATA* - AN AID TO REPLANNING



* ERBS AGSS

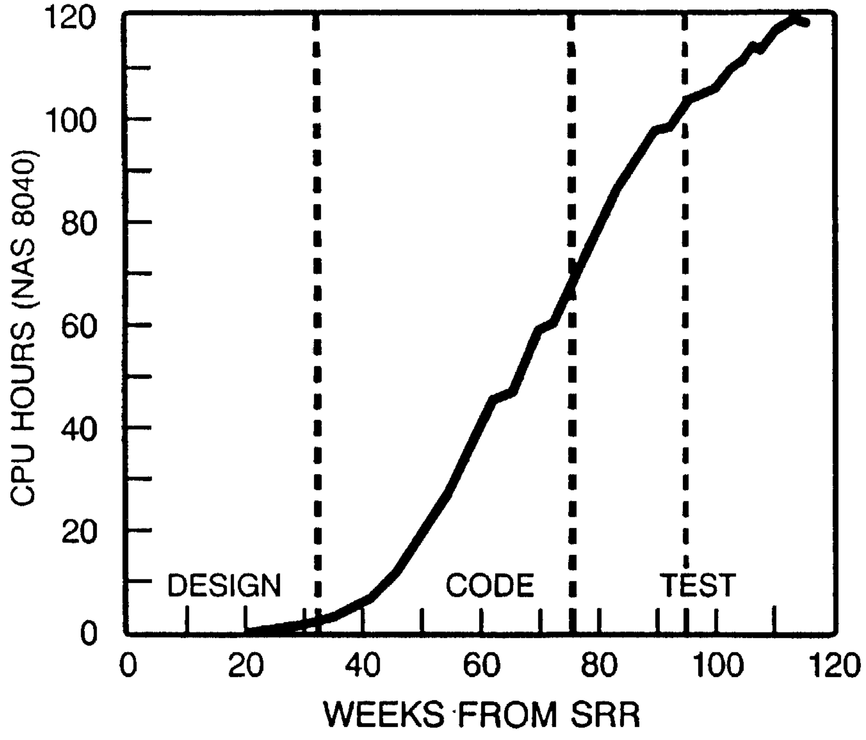
J217.018

VIEWGRAPH 13

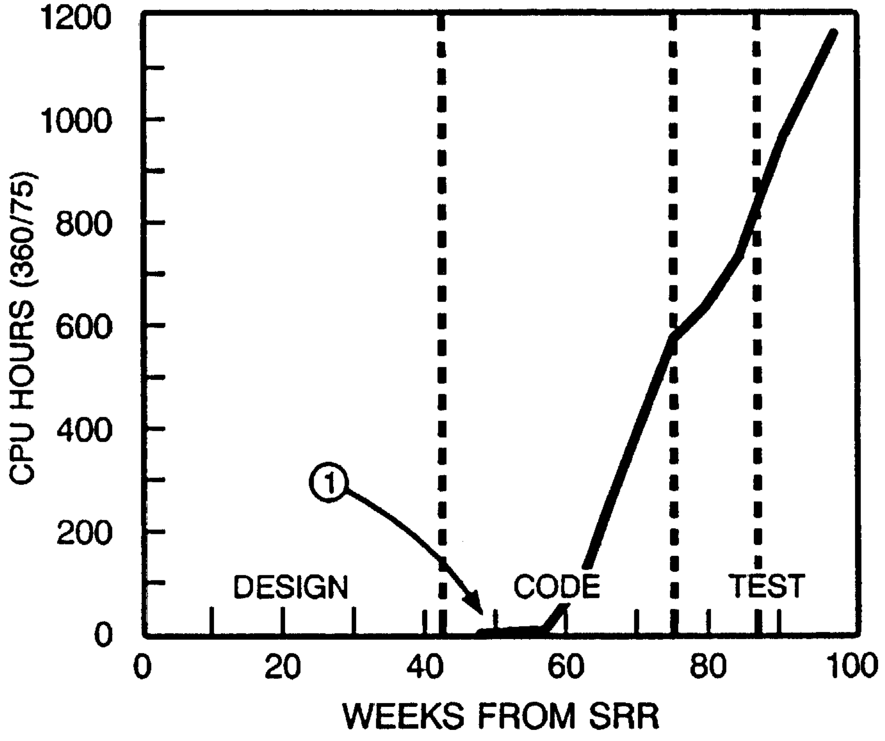
TRACKING COMPUTER USE

(REFLECTING REQUIREMENTS AND DESIGN STABILITY)

CUMULATIVE CPU DATA - PROJECT 1 (GOES)
(TYPICAL PROJECT IN SEL ENVIRONMENT)



CUMULATIVE CPU DATA - PROJECT 2 (ERBS)



① EXTENSIVE REQUIREMENTS ADDITIONS RESULT IN SIGNIFICANT REDESIGN - DELAY IN IMPLEMENTATION

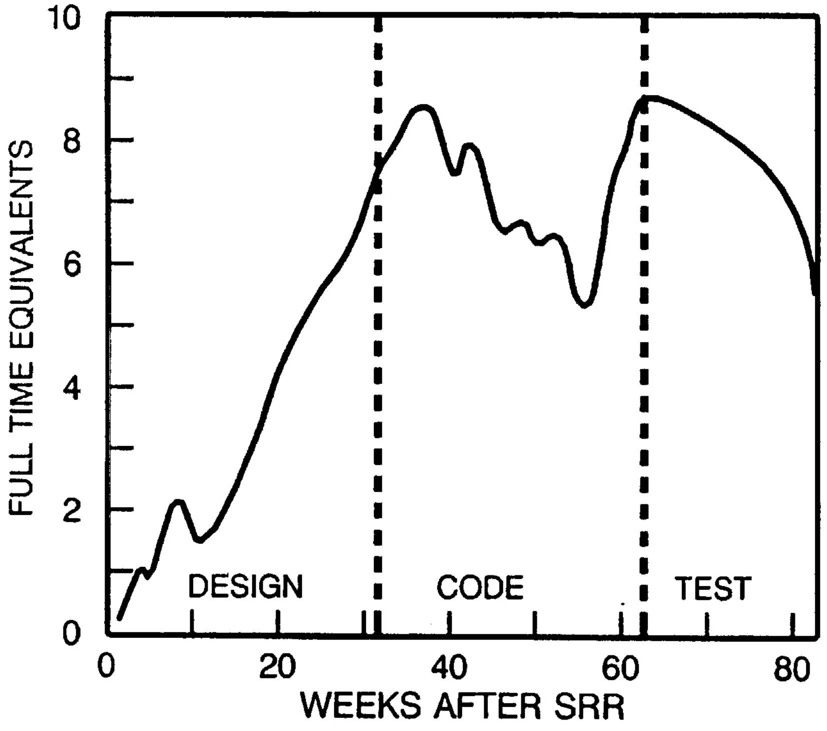
J217.019

VIEWGRAPH 14

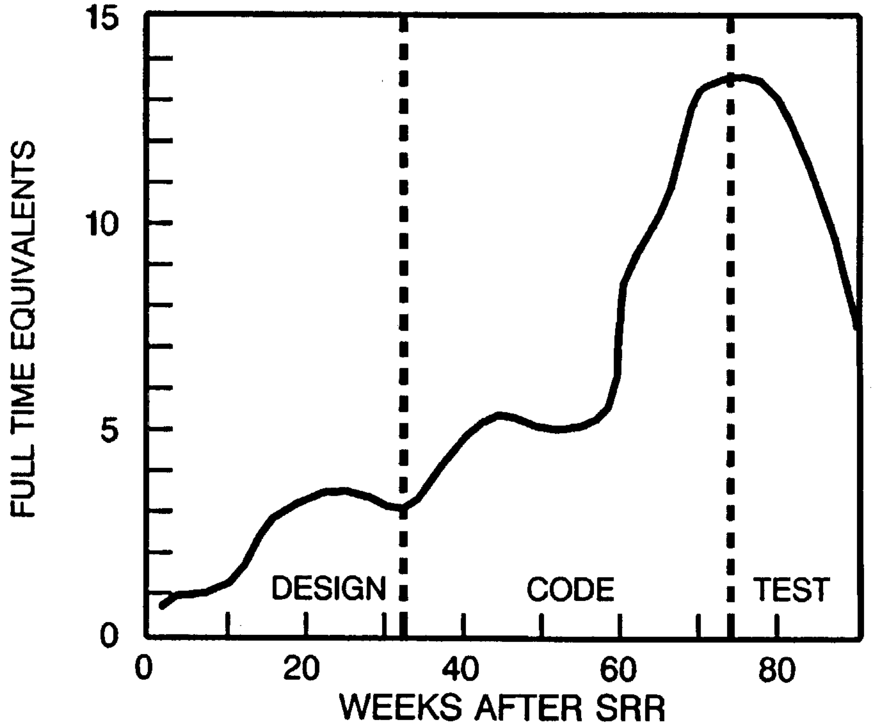
CHARACTERISTIC STAFFING PROFILES

(COMPARING 2 PROJECTS OF SIMILAR COMPLEXITY)

STAFF EFFORT/WEEK PROJECT 1 (DE-B)
(TYPICAL OF SEL PROJECTS)*



STAFF EFFORT/WEEK PROJECT 2 (DE-B)
(MYTHICAL MAN MONTH)



RELIABILITY & QUALITY OF PROJECT MAY BE REFLECTED IN STAFFING PROFILES

J217.020

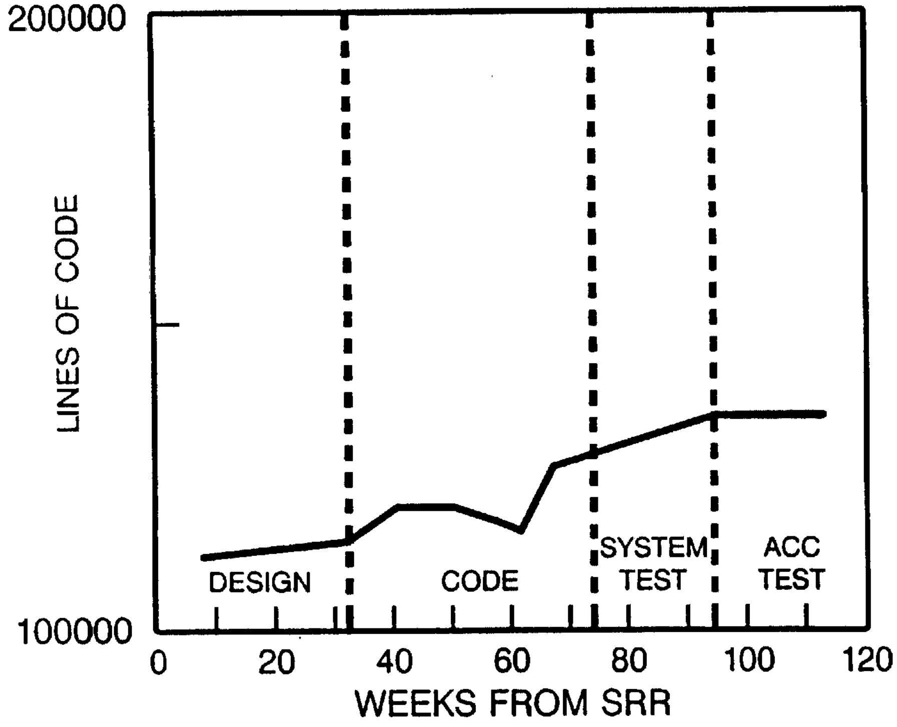
* REFERENCE - "ANALYZING MEDIUM SCALE SOFTWARE DEVELOPMENT" (BASILI, ZELKOWITZ)

VIEWGRAPH 15

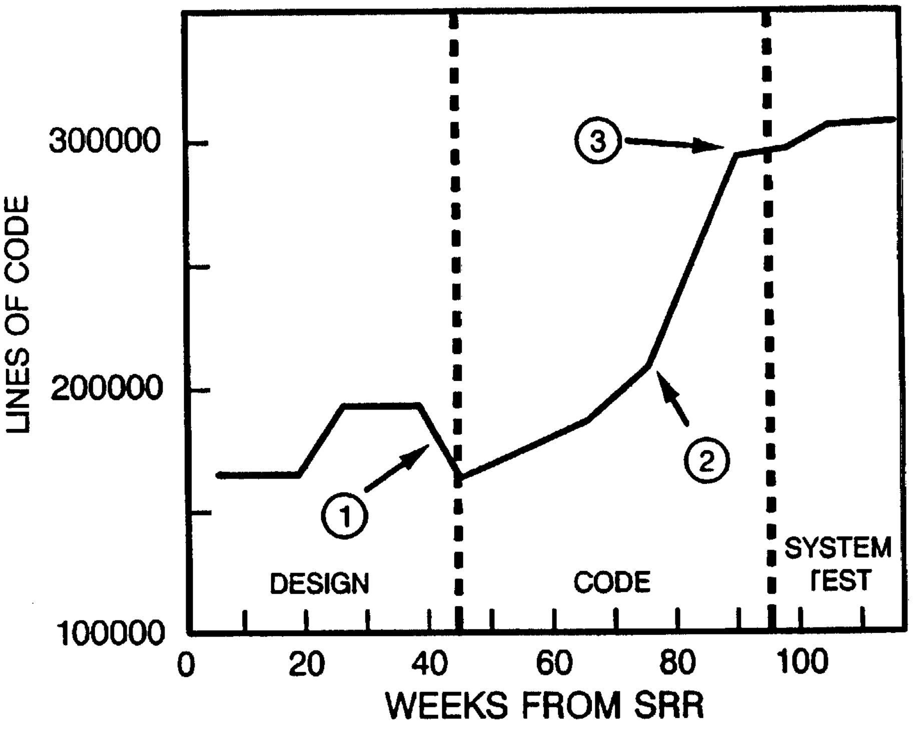
TRACKING FINAL SIZE ESTIMATES

(INDICATIONS OF “STABILITY”)

PROJECT 1 SIZE ESTIMATE
(TYPICAL FOR THIS ENVIRONMENT)



PROJECT 2 SIZE ESTIMATE



1. DELETION OF SOME REQUIREMENTS AFTER PDR
2. STRONG INDICATION - REVIEW NEEDED (NO ACTION TAKEN - TREND CONTINUES UNCONTROLLED)
3. DETAILED MANAGEMENT REVIEW (COST/SCHEDULES/REQUIREMENTS CHANGES/...; CONTROL FURTHER SPEC. CHANGES)

J217.021

VIEWGRAPH 16

SEL APPLICATION OF MEASUREMENT

3

PRODUCES FOUNDATION FOR

PLANNING FUTURE PROJECTS

J217.022

VIEWGRAPH 17

PLANNING AIDS FOR ONE ENVIRONMENT*

ESTIMATED PROJECT COST	EFFORT = 1.48 *KSLOC $^{.98}$	
EFFORT DISTRIBUTION	PRELIMINARY DESIGN	15%
	DETAILED DESIGN	17%
	CODE/TEST	26%
	SYSTEM TEST	23%
	ACCEPTANCE TEST	19%
COMPUTER UTILIZATION	NUM RUNS = 108 + 150 *(KSLOC)	
PAGES OF DOCUMENTATION	DOC = 34.7 (KSLOC $^{.93}$)	

*REFERENCES: "FINDING RELATIONSHIPS BETWEEN EFFORT AND OTHER VARIABLES IN THE SEL", BASILI, PANLILIO - YAP
"PROGRAMMING MEASUREMENT AND ESTIMATION IN THE SEL", BASILI, FREBURGER

J217.023

VIEWGRAPH 18

ADDITIONAL LOCAL PLANNING AIDS

Ada SOURCE CODE SIZE[1]

	FORTRAN LINES	Ada LINES
TOTAL PHYSICAL LINES	1	3
NON-COMMENT LINES	1	2.5
EXECUTABLE LINES	1	2
STATEMENTS	1	1

SAMPLE ADDITIONAL RELATIONSHIPS[2]

ERRORS	≈ # CHANGES/3.
FINAL CODE SIZE	≈ 1.4 * SIZE AT PDR
MAINTENANCE EFFORT	≈ 12% DEVELOPMENT EFFORT/YR

LOCALLY DETERMINED RELATIONSHIPS AND MODELS
KEY TO SUCCESSFUL PLANNING/MANAGING

1. REFERENCE: "EVALUATION OF Ada TECHNOLOGY IN A PRODUCTION SOFTWARE ENVIRONMENT" (MCGARRY, ESKER, QUIMBY)
2. BASED ON OBSERVATIONS/MEASURES OF S/W MANAGERS IN THE SEL

J217.024

VIEWGRAPH 19

SEL APPLICATION OF MEASUREMENT

④

PROVIDES RATIONALE

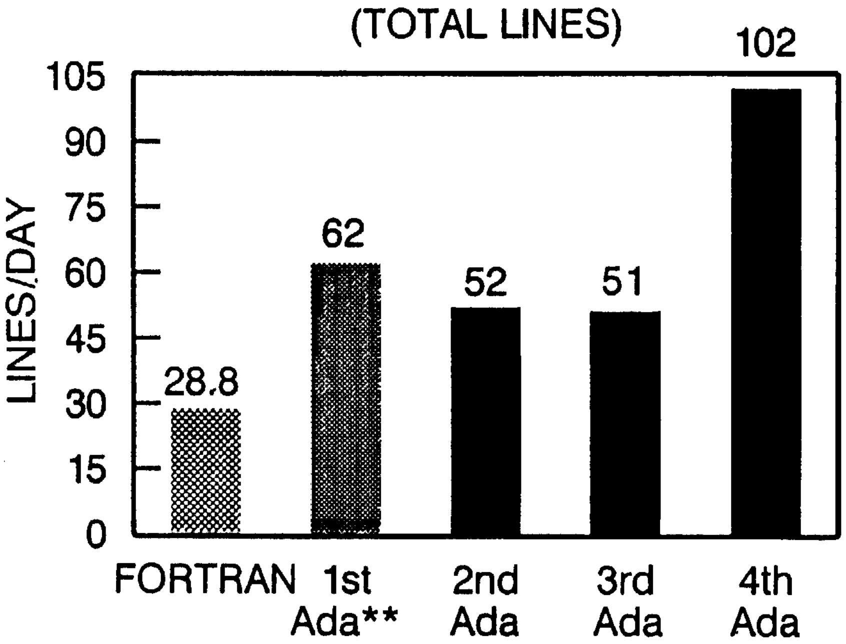
FOR

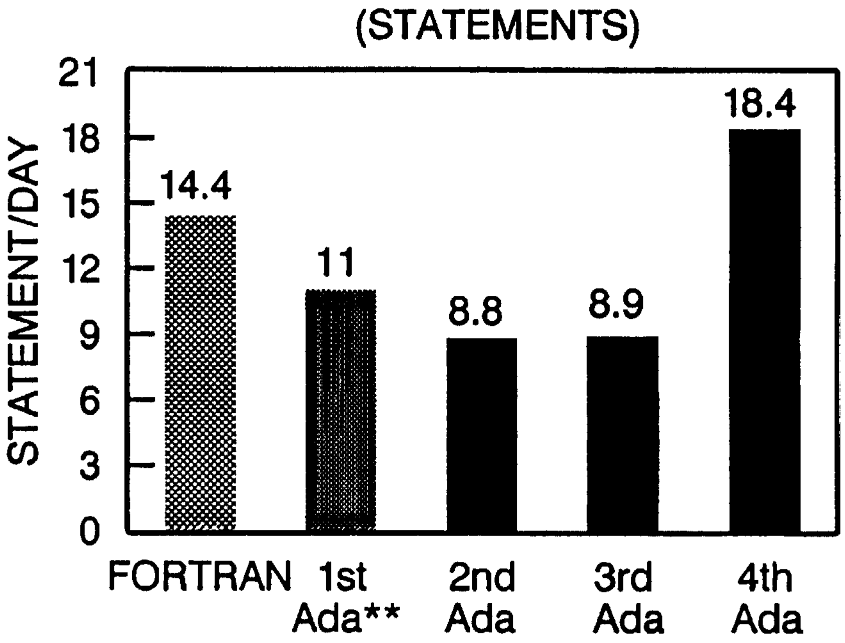
ADOPTING DEVELOPMENT PRACTICES/GUIDELINES

J217.025

VIEWGRAPH 20

IMPACTS OF Ada ON A PRODUCTION ENVIRONMENT*





PRODUCTIVITY WITH Ada NOT YET CLEAR
TRENDS SEEM POSITIVE OVER TIME

*BASED ON 7 Ada PROJECTS RANGING IN SIZE FROM 85 TO 160 KSLOC

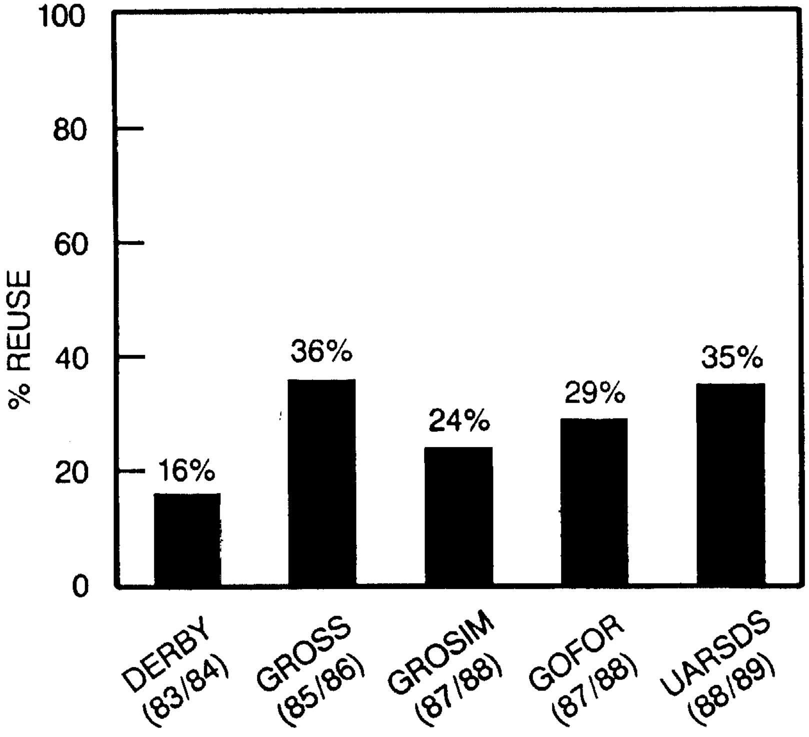
**FIRST Ada IS A SPECIAL STUDY PROJECT - NOT AN OPERATIONAL SYSTEM

J217.026

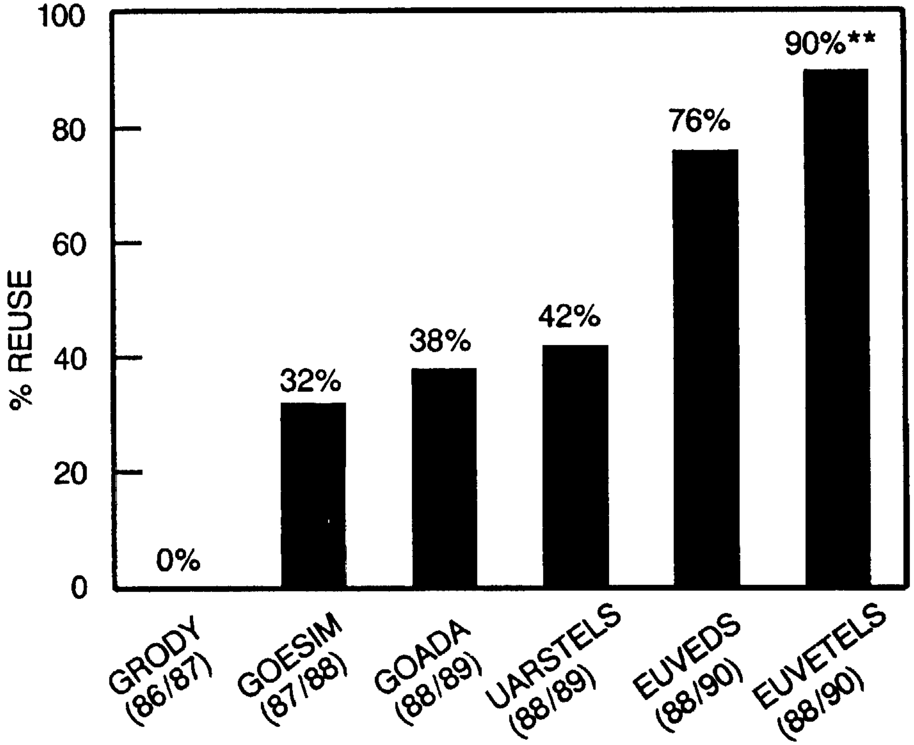
VIEWGRAPH 21

USE OF OBJECT ORIENTED DEVELOPMENT (OOD) LOOKING AT CODE REUSE

5 RECENT PROJECTS USING STRUCTURED ANALYSIS (NO SIGNIFICANT TRENDS)



5 PROJECTS USING OOD*



VERY SIGNIFICANT TRENDS FOR OOD SUPPORTING REUSE

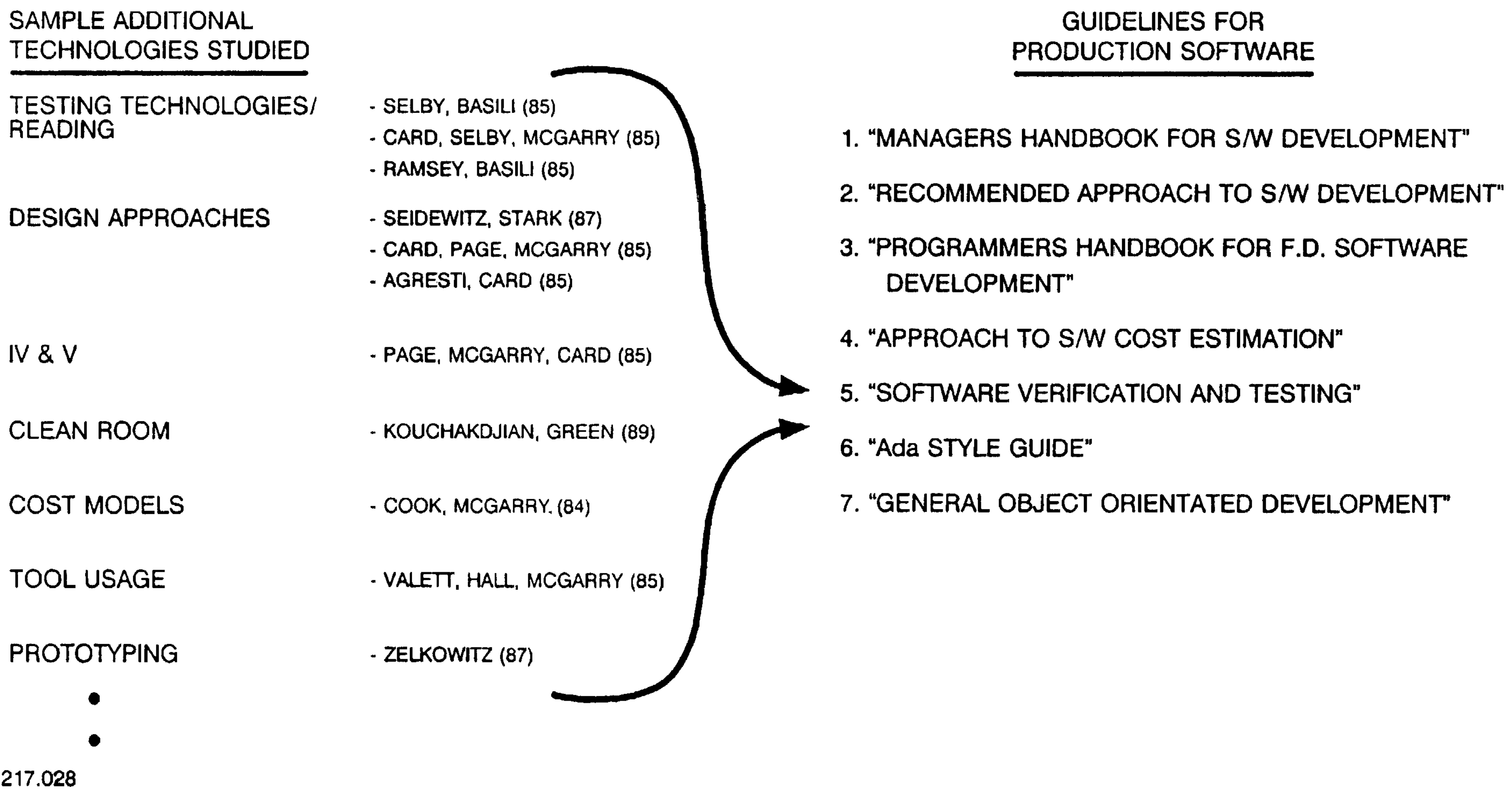
*ALL PROJECTS IN Ada

**EUVETELS SPECS WRITTEN AS A DERIVATION OF UARSTELS

J217.027

VIEWGRAPH 22

CLOSING THE LOOP
APPLYING MEASURED TECHNOLOGIES IN THE SEL



SAMPLE ADDITIONAL TECHNOLOGIES STUDIED	
TESTING TECHNOLOGIES/ READING	- SELBY, BASILI (85) - CARD, SELBY, MCGARRY (85) - RAMSEY, BASILI (85)
DESIGN APPROACHES	- SEIDEWITZ, STARK (87) - CARD, PAGE, MCGARRY (85) - AGRESTI, CARD (85)
IV & V	- PAGE, MCGARRY, CARD (85)
CLEAN ROOM	- KOUCHAKDJIAN, GREEN (89)
COST MODELS	- COOK, MCGARRY. (84)
TOOL USAGE	- VALETT, HALL, MCGARRY (85)
PROTOTYPING	- ZELKOWITZ (87)
•	
•	

GUIDELINES FOR PRODUCTION SOFTWARE

1. "MANAGERS HANDBOOK FOR S/W DEVELOPMENT"
2. "RECOMMENDED APPROACH TO S/W DEVELOPMENT"
3. "PROGRAMMERS HANDBOOK FOR F.D. SOFTWARE DEVELOPMENT"
4. "APPROACH TO S/W COST ESTIMATION"
5. "SOFTWARE VERIFICATION AND TESTING"
6. "Ada STYLE GUIDE"
7. "GENERAL OBJECT ORIENTATED DEVELOPMENT"

J217.028

VIEWGRAPH 23

Evaluation of the Cleanroom Methodology in the Software Engineering Laboratory

Ara Kouchakdjian (University of Maryland)
Scott Green (NASA/GSFC)
Victor Basili (University of Maryland)

[Slide 1]

Over the past two years, the Software Engineering Laboratory (SEL) has conducted an experiment using the Cleanroom software development methodology. The methodology is being used on an actual Flight Dynamics Division (FDD) project in order to evaluate the feasibility of Cleanroom in the environment. This presentation will first focus on a description of the methodology. After that, the experiment itself will be explained. Finally, some results will be shared and future work will be described.

[Slide 2]

The Cleanroom methodology was conceived by Dr. Harlan Mills, formerly at IBM–Systems Integration Division (IBM–SID), in the early 1980's. The goal of Cleanroom is to develop software that is 'right the first time.' Mills' contention is that the best tool for software development is the human mind. Unfortunately, it is also the most underutilized tool. The 'right the first time' goal is achieved by three activities. First, there is an emphasis on human discipline in program verification rather than computer aided program debugging. The concept behind this belief is that a high quality software product is built by solid design and development practices, not by debugging a mediocre product. This concept is facilitated by using a top down development approach, with a large number of builds. In this manner, a system is broken down into many small pieces, each of which can be solved and verified correctly, resulting in a high quality system. The second manner by which the 'right the first time' goal is ensured involves the complete separation of the development and test teams. Developers are not allowed to compile their own code, let alone unit test it. This forces developers to use good design and development techniques in order to produce a high quality product, since they do not have the luxury of testing the code. Third, software is developed with certifiable reliability, which is assessed in terms of Mean Time to Failure (MTTF). This approach, along with the top down development, allows the quality of the system to be continually assessed during the testing process. With Cleanroom, the emphasis is on error prevention, not error removal.

[Slide 3]

Development with the Cleanroom process is done at a desk or on a personal computer. Once again, developers cannot test the code, nor can they compile it. The developers read and review the code until they are convinced that the code is correct. At that point, they submit the code to the testers, who put it under configuration control, then compile, link and execute the code on the mainframe. Testers use a statistical testing approach, where test cases are generated according to the operational profile of the final system. When failures occur, the code is returned to the developers and corrected.

[Slide 4]

Cleanroom has been used previously at IBM and at the University of Maryland, with significant success in both environments. Surprisingly, there has been little additional use of Cleanroom in other environments. The Cleanroom experiment in the SEL is significantly different than the previous uses of Cleanroom. First, the organization is independent from those who conceived of the Cleanroom method. Second, the system being developed is much larger than the one developed in the controlled experiment at the University of Maryland, and is a production system. Finally, the FDD environment is one in which there is a large amount of change. The system specifications frequently change throughout the development lifecycle. This was a major area of concern as it is difficult to develop software 'right the first time' when the concept of 'right' may often be changing.

[Slide 5]

The primary reason for the SEL Cleanroom experiment was to possibly improve the way software is developed in the FDD. This includes improvements to both the process and the product. For example, the SEL was concerned about the large amount of time spent doing rework in the FDD environment. It is estimated that in this environment between 35% and 45% of the lifecycle effort is expended in rework activities. Comparable figures have also been reported at TRW. These activities include correcting errors, making changes, redesigning components, and implementing modifications to the specifications. There was hope that Cleanroom could help decrease the amount of rework done on projects.

In addition, the SEL wanted to apply, assess, refine and reapply the Cleanroom methodology as described by the Improvement Paradigm. With the description of the method now complete, the experiment can now be discussed.

[Slide 6]

The experiment is being conducted on an actual production system of approximately 33,000 lines of FORTRAN code. The staff was separated into development and test teams and spent approximately half their schedule on the project, as all personnel work on multiple projects. This was the first time that any of the staff had worked on the specific application, and, of course, this was the first application of Cleanroom by the personnel.

[Slide 7]

The project was completed over 22 months, from January of 1988 to November of 1989. At the present time, the subsystem is at the end of system test. A month of training served as preparation for the project. Training activities included related readings, a number of project meetings, and a one week tutorial on the Cleanroom methodology presented by Victor Basili of the University of Maryland and Michael Dyer and F. Terry Baker of IBM-SID. The focus of the tutorial was on previous experience with Cleanroom and a detailed description of the method.

When looking at the schedule, one notices an overlap between the coding and testing phases. This is possible because they are being done by two different teams, which allows the activities to be done in parallel. In addition, the top down development approach allows the developers to work on the second build of the system while the testers are testing the first. Each of the six builds contained approximately 5000 source lines of code, which is much smaller than typical build sizes in this environment.

[Slide 8]

The Cleanroom method was tailored to better fit the FDD environment. Typically, the FDD follows a waterfall approach to software development, with the development lifecycle divided into sequential phases. During the Cleanroom project, design, implementation and testing activities occurred simultaneously, although a top down development approach was still followed. In the FDD, the developers and testers are often the same, whereas the development and test teams were completely separated on the Cleanroom project. During the design phase, the Cleanroom developers improved and corrected the design as a team, rather than having one developer read another's program design language (PDL). A more thorough code reading process was also employed, where two developers would read the code written by the third. The code would be reread until the developers were convinced that the code had no remaining faults. This process replaced the typical code reading and unit testing done in this environment. Finally, the statistical testing approach was significantly different than the system and integration testing which is employed in the FDD.

Since the description of the experiment itself is now complete, there is now a context in which to view the results.

[Slide 9]

In terms of the distribution of effort in various activities during the life cycle, we see a notable difference between typical SEL projects and the Cleanroom project. Significantly more time was spent in design on the Cleanroom project than on typical SEL projects. Additionally, the effort distribution during the coding phase was also different. The coding phase consisted of two activities, writing code and reading code. Typically, 15% of the coding effort is expended reading code. On the Cleanroom experiment, over 50% of the coding effort was spent reading code. A different distribution was expected as the developers did not unit test their code, and relied heavily on the code reading process as the only means of verification. Overall, we see that the total effort distribution is significantly different.

[Slide 10]

When looking at the growth of the system with regard to both size and number of changes, there are notable trends. Code and changes began to appear later with the Cleanroom project, as more time was spent in design. The growth rate was also greater for the Cleanroom project, which was to be expected. The Cleanroom growth profiles are quite different than those associated with typical SEL projects.

[Slide 11]

For a comparison of computer usage, the Cleanroom experiment was viewed in relation to three recently completed projects in the FDD. Since the three systems were between three and seven times the size of the Cleanroom project, the figures for all systems were normalized by the respective sizes of the systems in order to form a common basis of comparison. The two areas of comparison were the number of computer runs (compiles, links and executions) and the number of CPU hours used. Overall, the Cleanroom project used between 70% and 90% fewer computer resources than these three typical SEL projects. Again, this was expected as developers were not allowed to compile or unit test their code.

[Slide 12]

Next, the error and changes rate were compared, along with project productivity. Error and change rates are tracked from when code comes under configuration control through the end of system test. With typical FDD projects, the code goes into the system library after it is unit tested. This is a later time than the Cleanroom project, which delivers code to the controlled library after it is code read. The error rate was found to be less than half the error rate on a typical SEL project. Of course, the acceptance test results will be the final gauge in understanding if Cleanroom actually leads to a lower error rate. The change rate was one third less on the Cleanroom project than on typical SEL projects. Finally, productivity is nearly 70% higher on this project when compared to other projects in this environment. The reasons for these impressive preliminary results must be further understood.

[Slide 13]

One of the original goals of the project was to decrease the rework effort. As previously stated, the error and change rates have decreased with the Cleanroom project. Additionally, the time to fix an error has also decreased. Typically, less than 60% of the errors are corrected in less than one hour. With the Cleanroom experiment, approximately 95% of all errors were corrected in less than one hour As these results seem to indicate a decrease in rework effort, one of the original goals of the Cleanroom experiment appears to have been satisfied.

[Slide 14]

Finally, the distribution of faults according to where they were found and corrected was viewed. This accounts for every fault found during designing, coding and testing. Over half of the total number of faults were found during the code reading phase.

Of those faults, less than 29% were found by both code readers. Since this means the vast majority of faults were found by only one reader, we are led to believe that two code readers were more effective than one on this project. During compilation, only 6% of the routines contained nonclerical faults. Most of the compilation faults were simple typographical errors. Overall, more than 87% of all faults were found before the code came under configuration control, and over 91% of all faults were found before the first test case was executed.

The fact that the SEL was able to use a version of the Cleanroom methodology in its environment, together with the early analysis, would lead one to conclude that the experiment was successful. Preliminary results, such as decreasing error and change rates, increased productivity, and a reduction in resource usage and rework efforts, are very impressive. Of course, much additional analysis remains.

[Slide 15]

The results must be better understood. Some results may be affected by extenuating circumstances, such as the quality of the staff working on the project. Final conclusions cannot be made until the subsystem is fully integrated and goes through its formal acceptance testing process. The first experience must also be tailored and packaged, as described in the Improvement Paradigm, so that additional experiments may be planned. Future experiments, coupled with what was learned in the first experiment, will allow the SEL to better understand and assess the Cleanroom methodology and its applicability in the FDD environment.

VIEWGRAPH MATERIALS

FOR THE

A. KOUCHAKDJIAN PRESENTATION

EVALUATION OF THE CLEANROOM METHODOLOGY IN THE SEL

ARA KOUCHAKDJIAN
UNIVERSITY OF MARYLAND

SCOTT GREEN
NASA/GSFC

VIC BASILI
UNIVERSITY OF MARYLAND

NOVEMBER 29, 1989

C55.001

SLIDE 1

CLEANROOM

A METHOD OF SOFTWARE DEVELOPMENT ATTRIBUTED TO HARLAN MILLS (IBM) THAT FOCUSES ON PRODUCING CORRECT SOFTWARE THE FIRST TIME, RESULTING IN THE PRODUCTION OF RELIABLE SOFTWARE PRODUCTS.

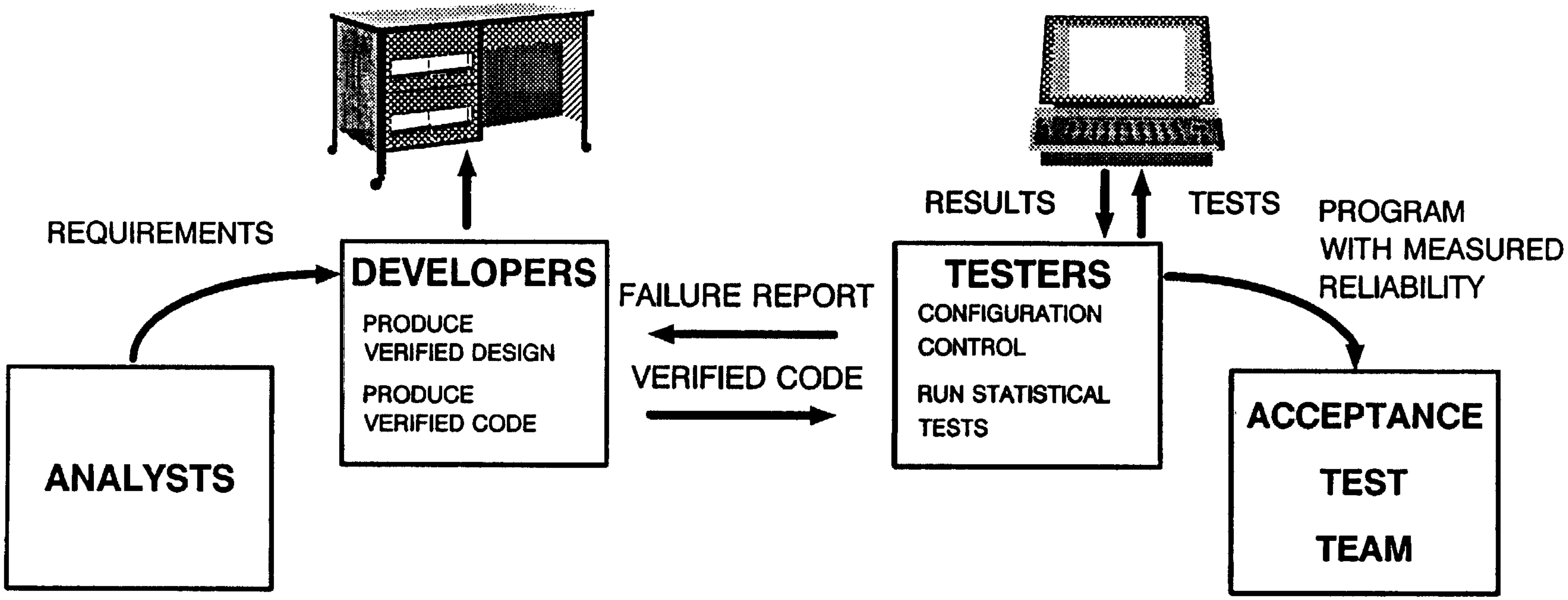
- EMPHASIZE HUMAN DISCIPLINE IN PROGRAM VERIFICATION RATHER THAN COMPUTER AIDED PROGRAM DEBUGGING
  (PROFESSIONAL EXCELLENCE)
- SEPARATE DEVELOPERS AND TESTERS
  (ORGANIZATIONAL CONTROL)
- PRODUCE SOFTWARE WITH CERTIFIABLE RELIABILITY
  (PRODUCT ASSESSMENT)

DEFECT PREVENTION *NOT* **DEFECT REMOVAL**

C55.002

SLIDE 2

THE CLEANROOM DEVELOPMENT PROCESS



- DEVELOPERS HAVE NO ACCESS TO MAINFRAME
  DESKTOP DEVELOPMENT ONLY
- TESTERS COMPILE, LINK AND EXECUTE CODE
  TOTAL CONFIGURATION CONTROL

C55.003

SLIDE 3

PREVIOUS CLEANROOM APPLICATIONS

IBM:

- 80 KELOC LANGUAGE PRODUCT
- HIGH QUALITY CODE - 3.4 ERRORS/KELOC
- HIGH PRODUCTIVITY - 740 LOC/STAFF MONTH
- WITHIN SCHEDULE AND BUDGET

UNIVERSITY OF MARYLAND:

- CONTROLLED EXPERIMENT (400 LEVEL COURSE)
- CLEANROOM GROUPS:
  - PASSED MORE TEST CASES
  - FULFILLED REQUIREMENTS BETTER
  - GENERATED LESS COMPLEX CODE
  - CODE HAD MORE COMMENTS

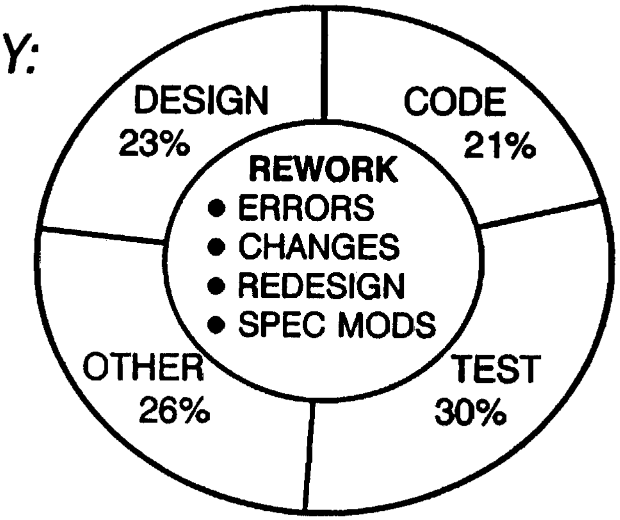
CLEANROOM HAS BEEN APPLIED SUCCESSFULLY IN OTHER ENVIRONMENTS

C55.004

SLIDE 4

REASONS FOR SEL EXPERIMENT

TYPICALLY:



(BASED ON HISTORY OF SEL SOFTWARE DEVELOPMENT)

PRODUCT IMPROVEMENT

- POTENTIAL TO SIGNIFICANTLY INCREASE QUALITY AND RELIABILITY OF SOFTWARE
- DECREASE TEST AND DEBUG TIME
- MINIMIZE REWORK EFFORT

PROCESS ASSESSMENT

- CHARACTERIZE CLEANROOM PROCESS AND PRODUCT
- COMPARE CLEANROOM PROCESS AND PRODUCT TO STANDARD SEL PROCESS AND PRODUCT
- EVALUATE, REFINE AND REAPPLY CLEANROOM METHODOLOGY

C55.006

SLIDE 5

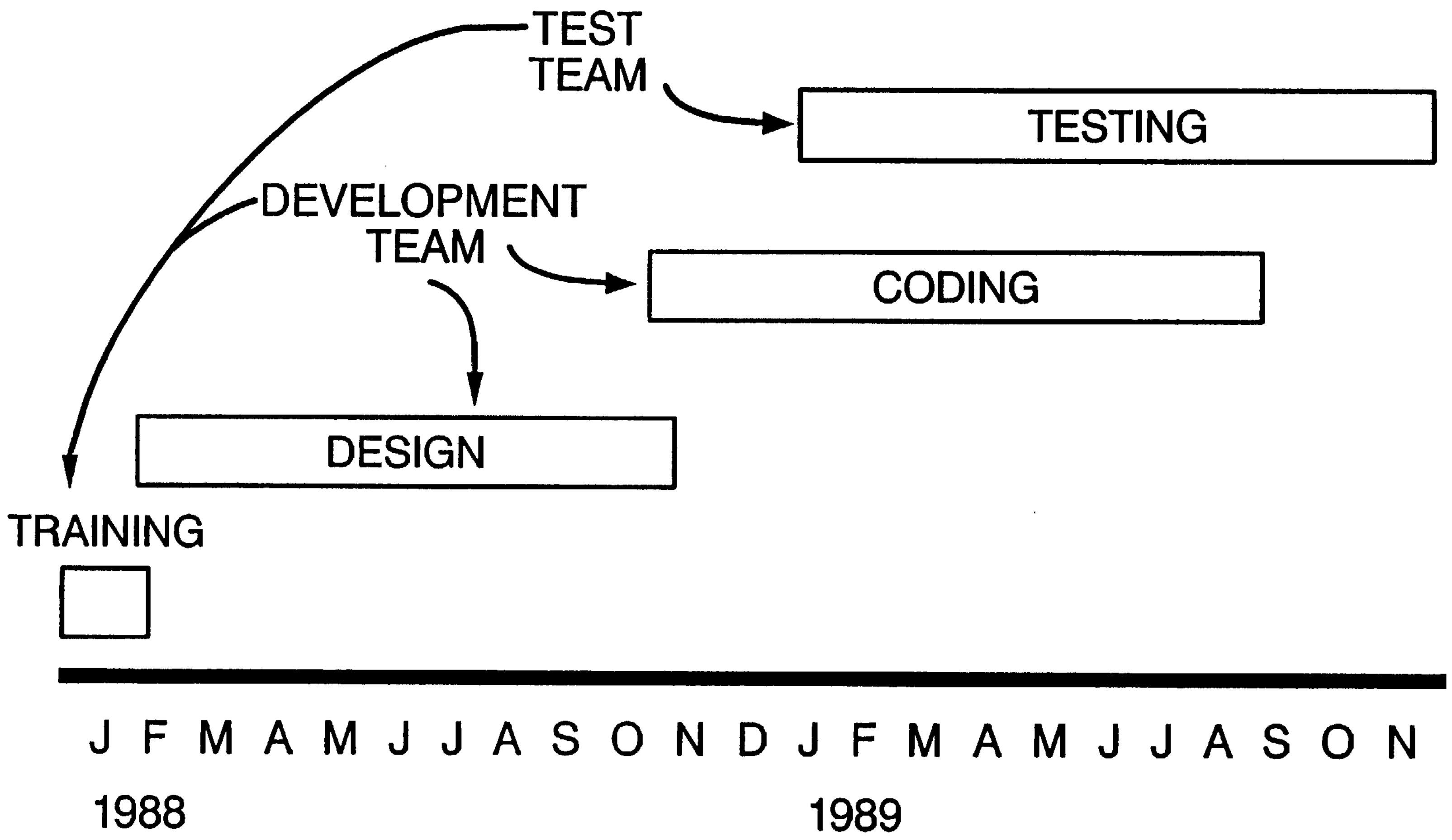
THE EXPERIMENT

SYSTEM	SUBSYSTEM OF A FLIGHT DYNAMICS SUPPORT SYSTEM
ENVIRONMENT	FORTRAN FOR AN IBM SYSTEM
TEAM	5 GSFC PERSONS
TEAM STRUCTURE	3 DEVELOPERS 2 TESTERS
SIZE	33 KSLOC
TEAM EXPERIENCE	FIRST USE OF CLEANROOM (TRAINING) FIRST EXPERIENCE WITH SPECIFIC APPLICATION 4 YEARS AVERAGE SOFTWARE EXPERIENCE
DATA COLLECTED	STANDARD SEL PROJECT DATA

C55.007

SLIDE 6

CLEANROOM PROJECT SCHEDULE



C55.008

SLIDE 7

PROCESS COMPARISONS

	ORGANIZATION	DESIGN	CODE	TEST
SEL CLEANROOM	*SEPARATE DEVELOPMENT AND TEST TEAMS*	*TEAM DESIGN REVIEWS*	*SEQUENTIAL REDUNDANT CODE READING*	*STATISTICAL TESTING*
TYPICAL SEL	*SINGLE DEVELOPMENT AND TEST TEAM*	*PDL READING*	*CODE READING AND UNIT TESTING*	*FUNCTIONAL TESTING*

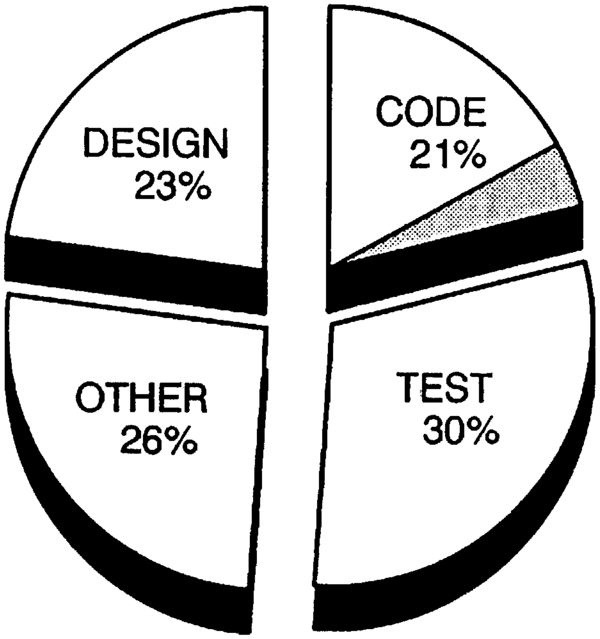
C55.005

SLIDE 8

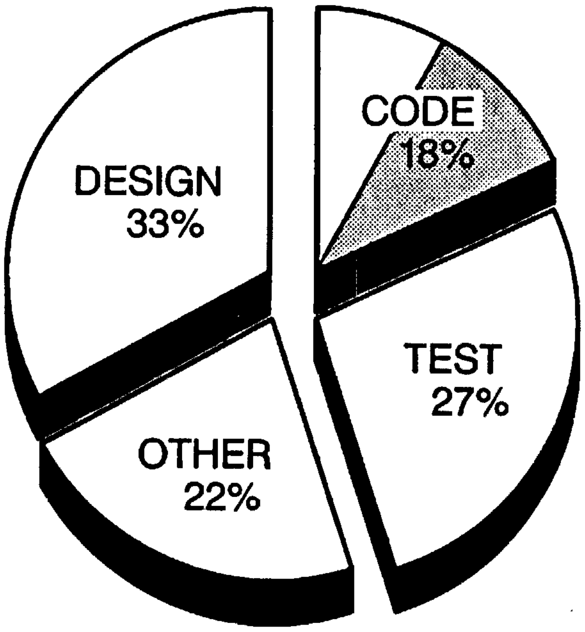
COMPARISON WITH TYPICAL SEL PROJECTS*

EFFORT

TYPICAL SEL EFFORT DISTRIBUTION



SEL CLEANROOM EFFORT DISTRIBUTION



- INCREASED DESIGN EFFORT WITH CLEANROOM
- CODE WRITING - CODE READING BREAKDOWN
  TYPICAL SEL: 85%-15%   SEL CLEANROOM: 48%-52%

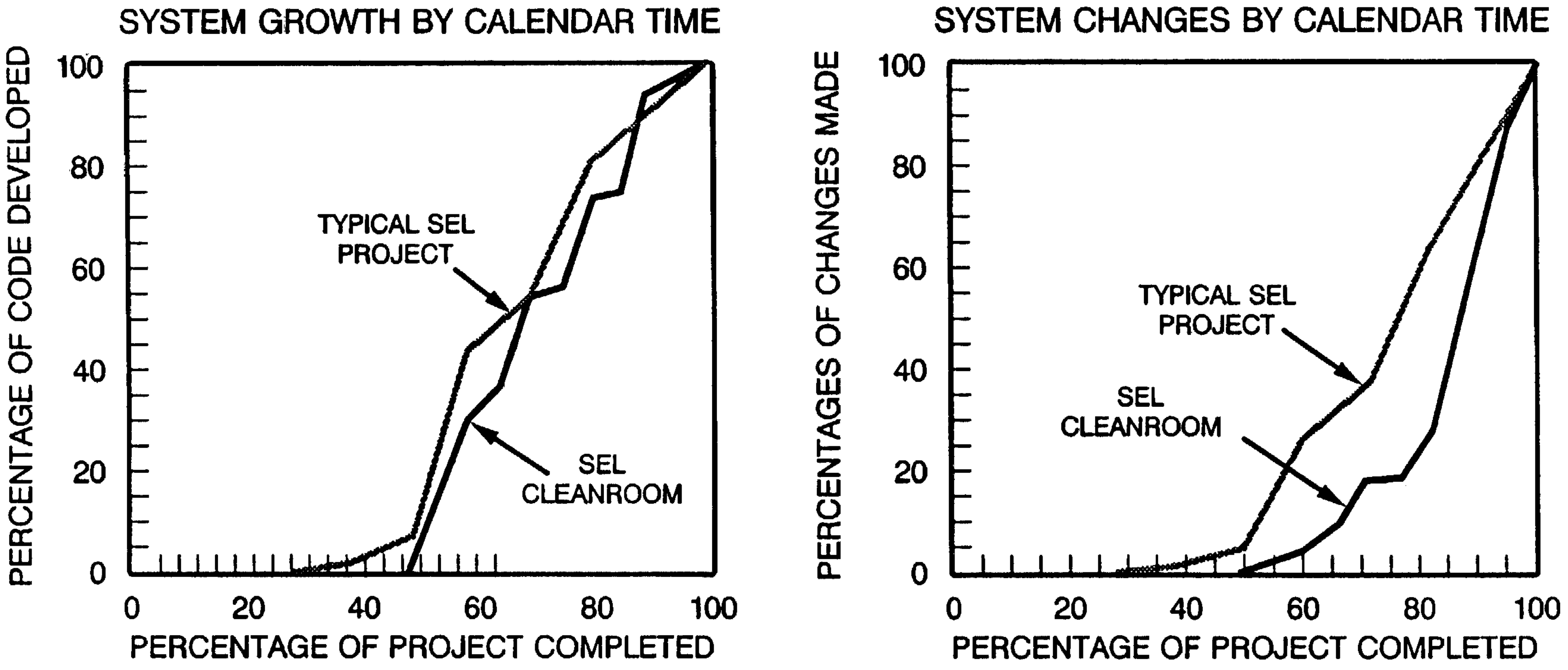
* THROUGH SYSTEM TEST

C55.009

SLIDE 9

COMPARISON WITH TYPICAL SEL PROJECTS*

SYSTEM GROWTH



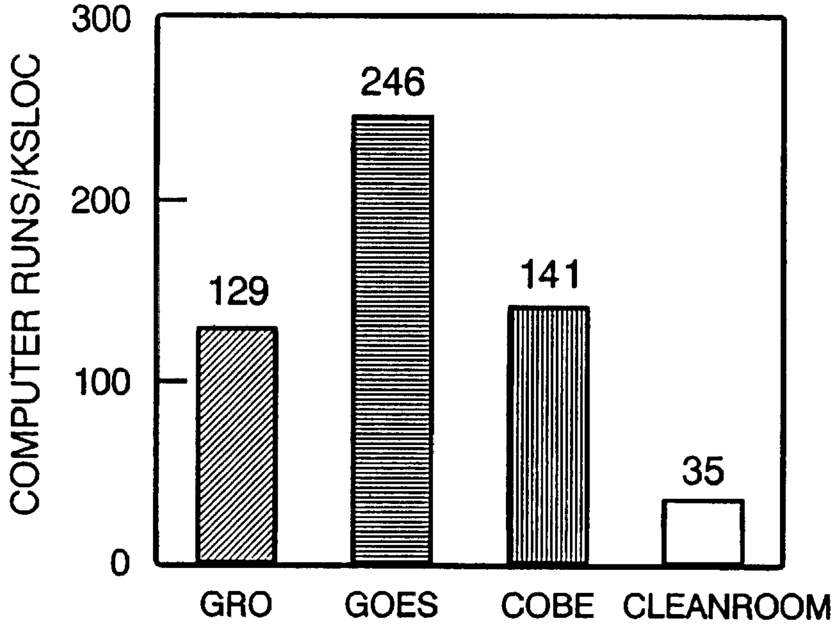
CODE AND CHANGES BEGIN TO APPEAR LATER, BUT GROW AT A FASTER RATE

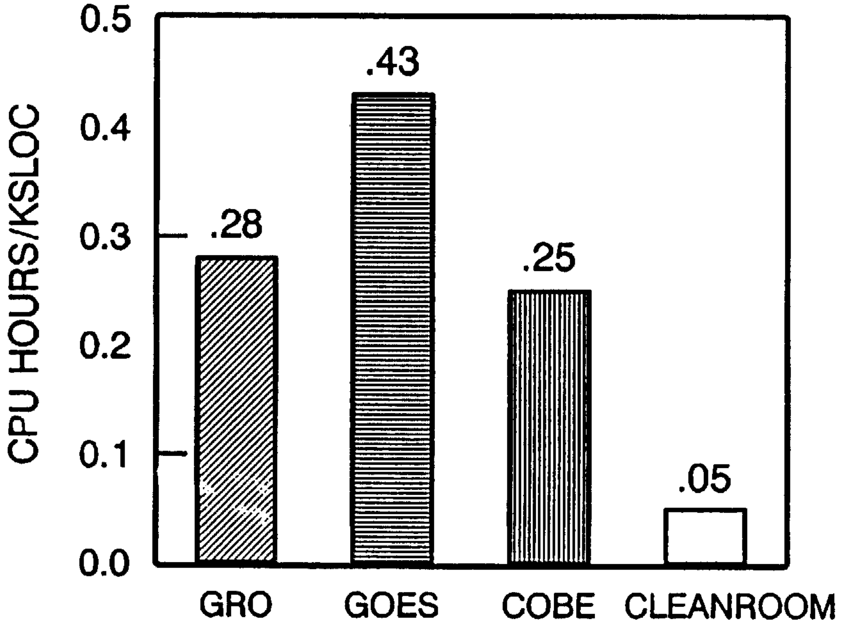
C55.016 *THROUGH SYSTEM TEST

SLIDE 10

COMPARISON WITH TYPICAL SEL PROJECTS*

COMPUTER USAGE





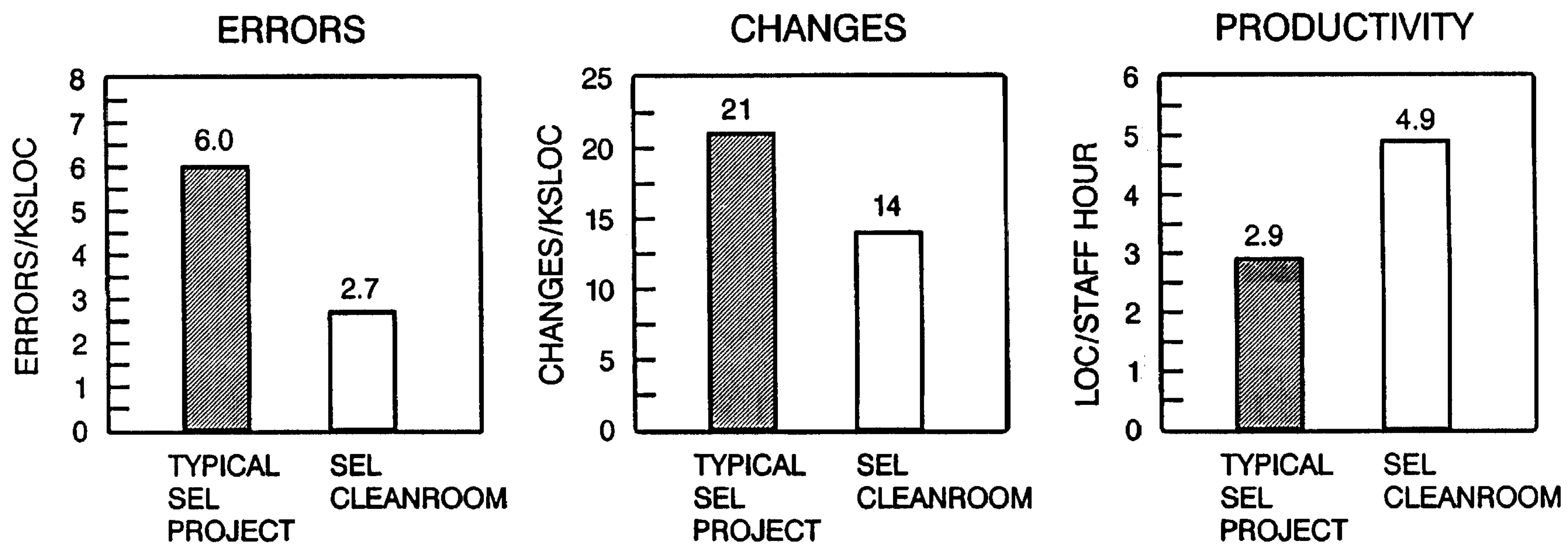
CLEANROOM USED LESS COMPUTER RESOURCES
THAN TYPICAL SEL PROJECTS

C55.010 *THROUGH SYSTEM TEST

SLIDE 11

COMPARISON WITH TYPICAL SEL PROJECTS*

ERRORS - CHANGES - PRODUCTIVITY



CLEANROOM COMPARES FAVORABLY WITH TYPICAL SEL PROJECTS

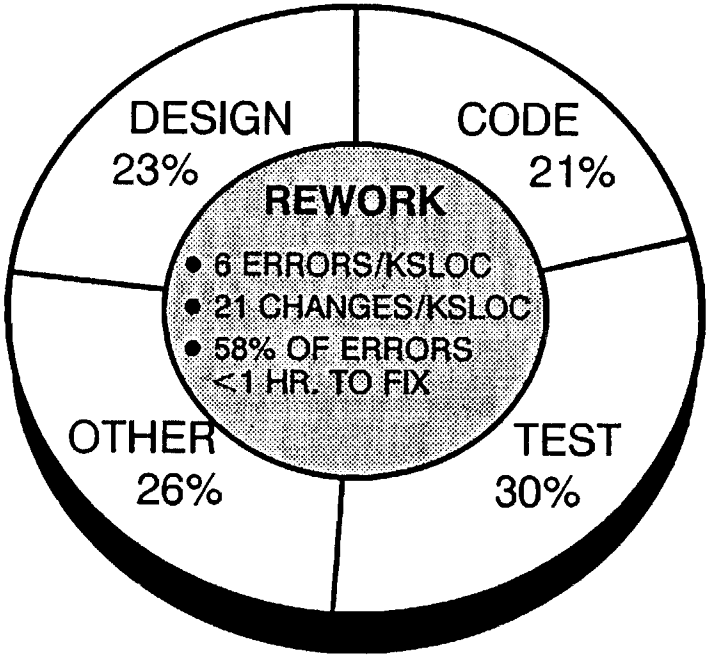
C55.011 *THROUGH SYSTEM TEST

SLIDE 12

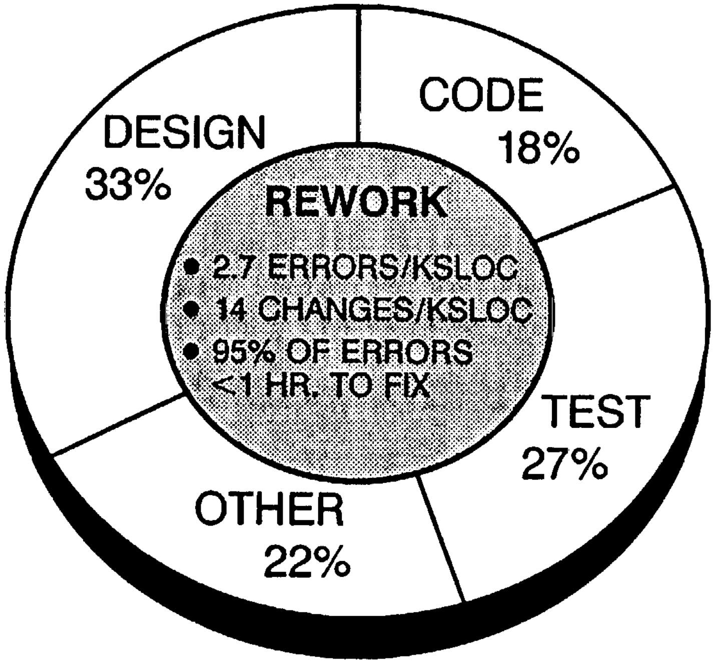
COMPARISON WITH TYPICAL SEL PROJECTS*

REWORK

TYPICAL SEL EFFORT DISTRIBUTION



SEL CLEANROOM EFFORT DISTRIBUTION

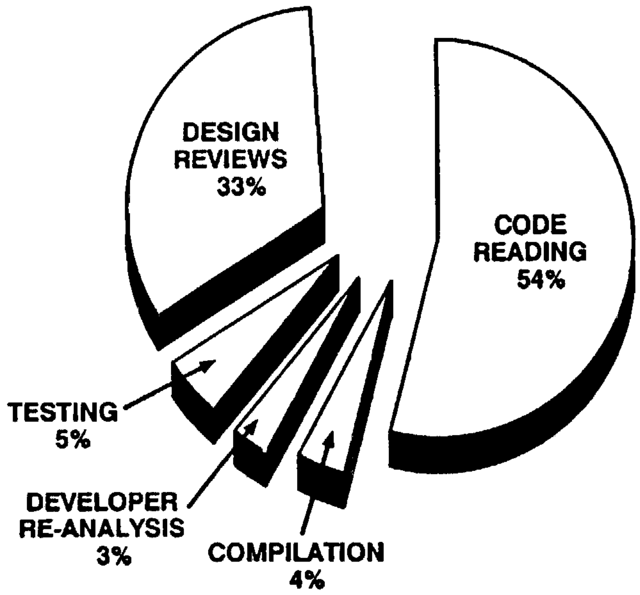


CLEANROOM REDUCED TYPICAL SEL REWORK EFFORT

C55.012 *THROUGH SYSTEM TEST

SLIDE 13

FAULT DISTRIBUTION BY QUALITY CONTROL ACTIVITY



- ONLY 29% OF FAULTS FOUND BY BOTH CODE READERS - TWO READERS ARE MORE EFFECTIVE, THAN ONE
- ONLY 6% OF MODULES CONTAINED NON-CLERICAL FAULTS
- 87% OF FAULTS FOUND BEFORE CODE CAME UNDER CONFIGURATION CONTROL
- 91% OF FAULTS FOUND BEFORE EXECUTION OF FIRST TEST CASE

OUR FIRST USE OF CLEANROOM SHOWS FAVORABLE MEASURES.

C55.013

SLIDE 14

FAULT DISTRIBUTION BY QUALITY CONTROL ACTIVITY

ACTIVITY

FAULT TYPE	DESIGN REVIEWS	CODE READING	COMPILATION	DEVELOPER RE-ANALYSIS	TESTING	TOTALS:
FORTRAN SYNTAX	0%	4%	100%	0%	0%	6%
CONTROL FLOW	20%	8%	0%	5%	8%	12%
INTERFACE	24%	17%	0%	45%	33%	20%
DATA INITIALIZATION	1%	5%	0%	8%	5%	4%
DATA DECLARATION	45%	19%	0%	5%	8%	26%
DATA USE	0%	32%	0%	32%	25%	20%
COMPUTATION	10%	9%	0%	3%	16%	9%
DISPLAYS	0%	6%	0%	2%	5%	3%
TOTAL NUMBER OF FAULTS:	542*	883	65	56	80	1626
PERCENT OF TOTAL FAULTS FOUND:	33%	54%	4%	3%	5%	

* - PROJECTED

C55.014

SLIDE 15

A. Kouchakdjian
Univ. of MD

CLEANROOM AND THE SEL: WHAT'S NEXT?

- COMPLETE CURRENT ANALYSIS
- TAILOR PROCESS TO BETTER FIT SEL ENVIRONMENT
  - DESIGN APPROACH
  - TESTING APPROACH
- REAPPLY TO ADDITIONAL SEL PRODUCTION EFFORTS
- REASSESS

SESSION 2 — METHODOLOGIES

M. S. Deutsch, Hughes Aircraft Co.

B. I. Blum, APL

H. D. Rombach, University of Maryland

PREDICTING PROJECT SUCCESS FROM THE SOFTWARE PROJECT MANAGEMENT PROCESS: AN EXPLORATORY ANALYSIS

Michael S. Deutsch
Hughes Aircraft Company
c/o Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Sponsored by U.S. Department of Defense

Abstract

The paucity of significant empirical data relating the software management process to quantitative project performance is the motivation for this study. A conceptual causal model characterizes the factors of adversity that may be present on a project along with the factors of management skills that are available to neutralize or overcome the adversity; the residual effect, called net turbulence, is hypothesized to relate quantitatively to project business and technical performance. An informal exploratory data analysis on 24 projects has been undertaken to determine the feasibility of the conceptual model and to identify more precise hypotheses for more formal study. The non-parametric coefficients of correlation for net turbulence and both project technical and business performance are 0.65, suggesting that the basic hypothesis of this model is feasible. Other interesting relationships involving risk management, project adversity, business constraints, and precision of technical scope definition deriving from the exploratory analysis are discussed.

BACKGROUND

Large contemporary R&D engineering projects for a wide variety of systems such as communications, process control, command and control, large scale data retrieval, and military applications are becoming increasingly software intensive, challenging human capacity to manage resulting intellectual complexity. The relative immaturity of the software engineering discipline has injected new uncertainties in human, business, and technical variables into these projects. There is relatively little empirical basis beyond the experience of individual managers to connect models of software project management with actual levels of success achieved. The approach to managing large scale intellectual efforts involving, perhaps, hundreds of people on software intensive systems has been based almost entirely on theory-based, anecdotal, or single-case study considerations rather than on any systematic empirical investigation into what factors actually contribute to positive software project performance. This paper describes an empirical study that addresses this gap.

A conceptual model of the software project management process is set forth that is asserted to relate to actual project performance. The scope of this paper is to: 1) describe this model; and 2) present an exploratory investigation that seeks to establish the feasibility of the conceptual model and sharpen its associated hypotheses. The longer range goal is to evaluate the predictive validity of the software management process on project performance through prospective observations from ongoing projects; but this will occur over a number of years. At present, the more modest and practical goal is to determine this feasibility based upon concurrent validity of retrospective data from completed projects. The exploratory feasibility analysis is based upon data from 24 projects that was informally collected; this exploratory analysis is in itself encouraging and interesting.

The intent of this investigation is to identify factors that discriminate between successes and non-successes on software projects. By looking backward from a successful result, a broader view of the management and technical actions that achieved the success can be viewed and contrasted against those from less successful projects. Unfortunately, measuring success in the traditional dimensions -- technical, schedule, and cost performance -- is frequently an inaccurate gauge by itself. The novelty content and technology demand of large, software intensive systems frequently yield results that are less than full expectations (in the traditional dimensions), yet many projects are considered successful nonetheless especially when an adverse and difficult project situation is at least partially overcome. This study probes into these further aspects of success by characterizing the factors of adversity that may be present in the project environment and the factors of management skills that may be put forth to manage and overcome this adversity. These are then related to both project technical and cost/schedule performance factors.

This empirical study is outlined in the following paragraphs by recounting related studies, defining the conceptual model and its components, delineating the causal hypotheses associated with the model, and summarizing an exploratory data analysis of 24 projects.

RELATED STUDIES

Practically all previous empirical research and investigation into project success factors have embraced a broad non-specific scope of general project situations such as construction, equipment, studies, services, or testing. Only a small subset of these studied projects appear to be of an R&D nature. Even less attention has been attributed to software projects.

The most prolific recent researchers in this field have been Pinto and Slevin. Among their contributions has been the development of a project implementation profile [1,2], a validated questionnaire tool for probing project success factors based upon a ten factor management process model. Schultz, Slevin, and Pinto have documented [3] a sample of five attempts by different researchers to determine critical success factors from which it is possible to discern some general factors. Pinto and Prescott [4] have further examined a set of five basic hypotheses embracing the ten factor project implementation profile against 408 projects from the manufacturing and service sectors; the results indicate that the relative importance of several of the critical factors change significantly over the project life cycle. Murphy, Baker, and Fisher [5,6] have studied 646 projects, primarily manufacturing and construction, illuminating those positive determinants and those negative determinants that are necessary to be encouraged and discouraged respectively to achieve potential success. The classes of projects involved in these studies render the results interesting, but they are applicable to R&D software intensive projects only in the most general sense.

One significant investigation of success factors for software specific projects is the study by Curtis, Krasner, and Iscoe [7] of 17 large client projects of the Microelectronics and Computer Technology Corporation. Their findings, concluded from an interview process, focus on the richness of application domain knowledge, fluctuating and conflicting requirements, and communications bottlenecks as the factors most influential on success. Another study of partial pertinence is the methodology for identifying critical success factors for management information systems developed by Boynton and Zmud [8].

A related area of activity over the past decade has been the development, maturation, and practical usage of parametric cost estimation models for software projects. These models address project success factors in a limited sense by predicting labor effort, schedule, and productivity based upon inputs of project size and characteristics. The models and associated investigations conceived by Boehm [9], Jensen [10], and Vosburgh, Curtis, Wolverton, et al [11] are representative of this work.

None of the cited related studies have attempted to broadly relate success factors in the software management process to quantitative project performance. This is a major goal of the study described in this paper.

CONCEPTUAL MODEL OF PROJECT SUCCESS

A hypothetical model relating project performance to the project management process was conceived by the author and refined as a result of consultations with colleagues in industry and government. The major thesis of this model is that project performance can be roughly predicted based upon how effectively the "power" of the management process cancels out the adverse attributes of the project; the residual of the cancellation effect between the management power and project adversity is referred to as the "net turbulence" of the project. The structure of this hypothetical model is displayed on Figure 1. The major hypothesis of the model is that the predictive measure, net turbulence, should be strongly correlated with the dependent measures of project success, technical and business performance.

The model is intended to depict the collective behavior of the three major parties who collaborate during a software system development: the eventual user of the system, the customer who financially sponsors the development, and the contractor who performs the system development. These three constituencies may be separate agencies, or they may belong to the same company or organization. Whatever the organizational configuration, the three roles are invariably identifiable.

Project adversity represents "facts of life" over which the three parties have, at best, a secondary level of control. Management power, on the other hand, symbolizes those factors where a primary level of control exists within the three parties at the project management level. The more detailed causal relationships between these variables reflecting the formation of the net turbulence parameter is presented shortly.

The factors associated with technical performance, business performance, project adversity, and management power are delineated on the figure and defined below.

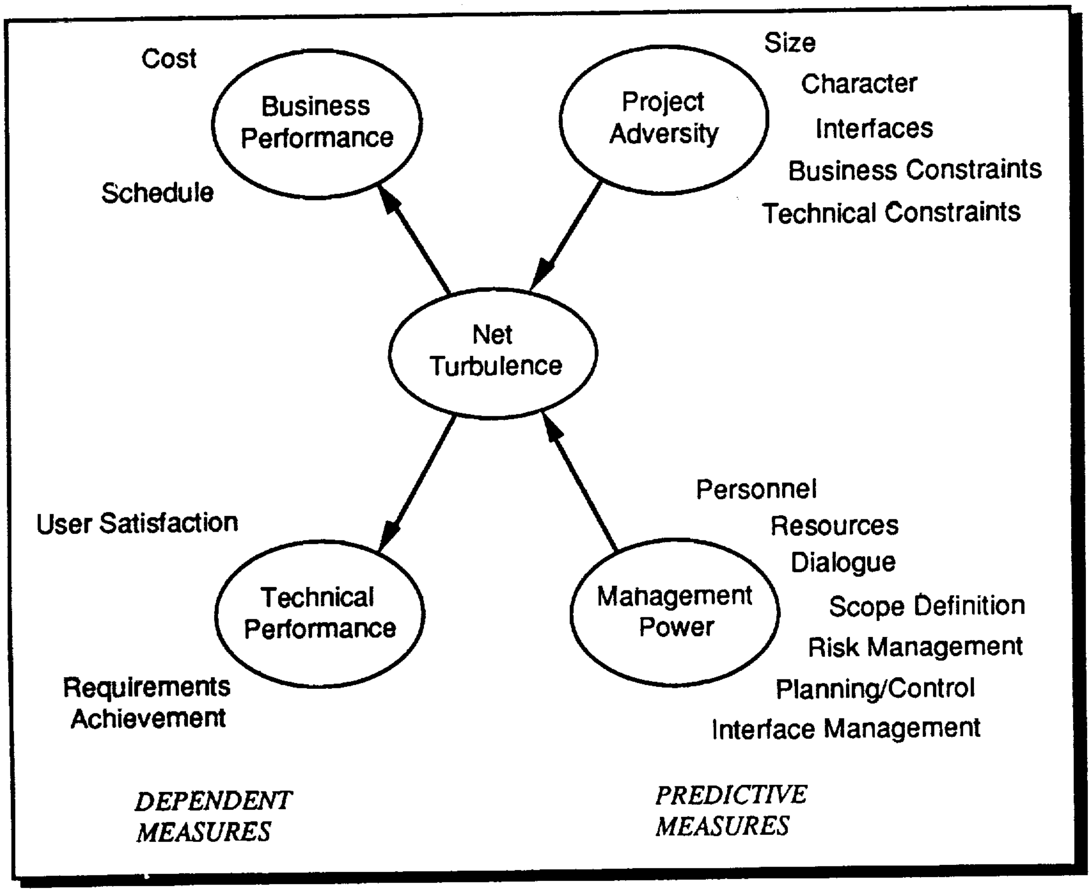


FIGURE 1: HYPOTHETICAL MODEL OF PROJECT SUCCESS

Technical Performance Factor Definitions

1. *User satisfaction.* The degree that users of the system were satisfied by system performance.
2. *Requirements achievement.* The degree that the specified functional, performance, external interface, operational scenario, and quality requirements were satisfied.

Business Performance Factor Definitions

1. *Cost performance.* The percentage variance between projected and actual costs.
2. *Schedule performance.* The degree that key schedule milestones were achieved.

Project Adversity Factor Definitions

1. *Project size and character.* The magnitude of the system product developed and its attributes that reflect internal difficulty and complexity.
2. *External interface adversity.* The attributes of the system that reflect complexity of interactions with the surrounding external environment.

3. *Business constraints.* The realism of the cost and schedule budgets for the project.
4. *Technical constraints.* Maturity and accessibility of the technology and process available to accomplish project tasks.

Management Power Factor Definitions

1. *Personnel resources.* Quality and retention of personnel across the project phases.
2. *Physical/technical resources.* Quality of the discretionary physical and technical resources assigned to the project.
3. *User/customer/contractor dialogue.* The degree and frequency of the mechanisms that the three parties used to conduct an on-going collaboration.
4. *Technical scope definition.* Clarity, scope, and stability of technical requirements.
5. *Strategic risk management and planning.* The scope of strategic planning measures for life cycle planning and risk reduction.
6. *Project planning/control.* The scope of tactical measures during project implementation for business, technical, and risk visibility and control.
7. *External interface activities.* Provision of appropriate activities and process steps for interactions with elements external to the system.

THE MANAGEMENT POWER SUB-MODEL

For heuristic purposes, a sub-model of the management process was constructed based on the seven management power factors defined above. This sub-model is shown on Figure 2. Conceptually, these factors or major activities tend to occur in a certain order while controlling the technical development. They embrace the traditional management functions of planning, organizing, staffing, directing, and controlling. It is recognized that some concurrency and iteration are present in the general pattern indicated on the diagram. Significant revisiting of previous plans and baselines may occur at each major activity as a result of a continuing dialogue between the user, customer, and contractor. Breakdowns in this communication process have been a major cause of unfulfilled project goals [6]. Paramount to this management process is the influence of this dialogue in temporally adjusting the definition of the technical scope as the project's needs become better understood. It can be argued, on an anecdotal basis, that misdefined technical scope is a major source of risk for software intensive systems [12].

Each of the seven management power factors are described below in more detail. The significance of the factor is discussed and a number of considerations and issues component to that factor are raised. These considerations and issues represent questions that project management should be analyzing beginning with project conceptualization to judge whether the power of the management process can overcome the project's adversity. Each question has a scale of discrete responses, not indicated here, that was included in the informal questionnaire used to collect exploratory data.

Strategic Risk Management/Planning Factor

The initial step of management planning should entail selection of a life cycle plan of phases appropriate to the risk and adversity level of the project including, for example, consideration of risk reduction measures such as concept exploration phases and/or prototyping. This level of planning may occur before there is a commitment to project implementation.

Considerations:

- Which risk reduction measures were included in the project life-cycle before commitment to full-scale development?
- Is a life cycle cost analysis part of the scope of work?
- Are user operational staff levels included in system tradeoffs?
- Is a design-to-cost approach part of the system tradeoffs?

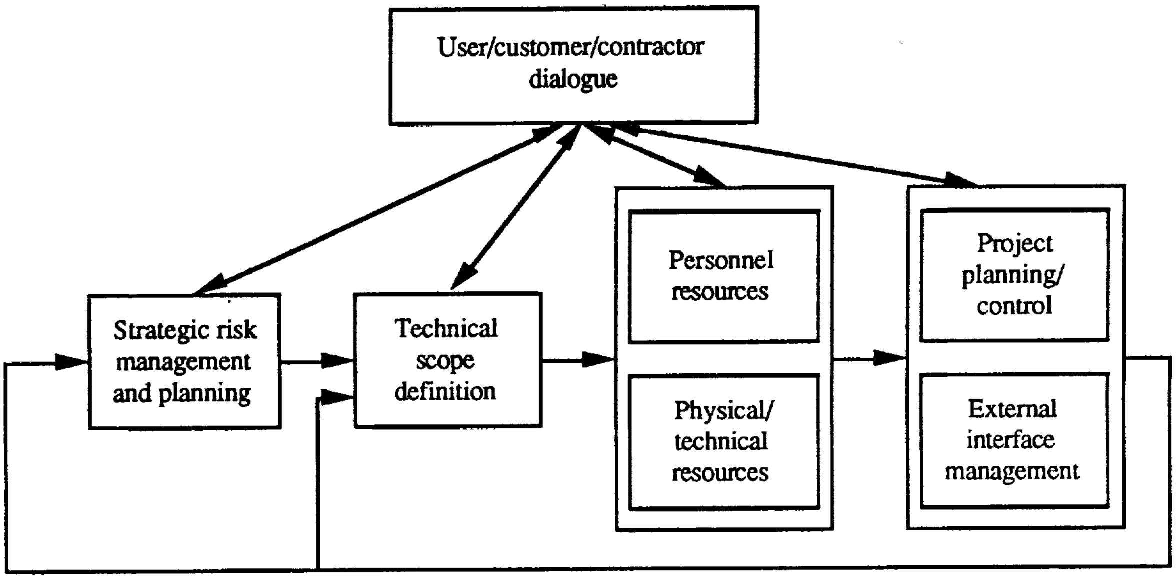

User/customer/contractor dialogue
Strategic risk management and planning
Technical scope definition
Personnel resources
Physical/ technical resources
Project planning/ control
External interface management

FIGURE 2: MANAGEMENT PROCESS MODEL

Technical Scope Definition Factor

Although the technical scope definition may be time varying, there should exist a baselined consensus between user, customer, and contractor that evolves in a controlled way.

Considerations

- How well are functional requirements specified?
- How well are performance characteristics specified?
- How well are operational scenarios specified?
- How well are system qualification requirements specified?

- How well are operational personnel and post deployment support requirements specified?
- How well are computer-human interface requirements specified?
- How well are quality requirements specified?
- Are requirements under change control?

Personnel Resources Factor

A clear issue here is the initial selection of personnel with the right blend of applications expertise and functional disciplines. Selection of personnel invariably requires a negotiating process. Neglect of this factor defaults to use of people who are conveniently available regardless of their value to the project [1]. Another major consideration is the retention of a skilled cadre of personnel who remain on the project through testing and at least initial post deployment maintenance.

Considerations:

- To what degree are personnel experienced in the required functional disciplines available?
- To what degree are personnel experienced in the required applications areas available?
- How skillful is the contractor project manager?
- How skillful is the customer project manager?
- How skillful is the user representative?
- What is the skill level of the engineers/application experts who remain on the project through testing and transition to operations?
- How sufficient is the engineering and application expertise of the initial post deployment maintenance team?

Physical/Technical Resources Factor

These are the resources that constitute the environment that surrounds the project development. Management usually has a primary level of control of selection of these resources.

Considerations:

- How mature is the selected computing hardware and support software?
- How mature are the software engineering support tools?
- To what degree are the needed facilities available to the project?

Project Planning and Control Factor

The monitoring, feedback, and risk control mechanisms in this factor give management the visibility into evolving problems and ability to oversee corrective actions.

Considerations:

- Is there a project function (e.g., a system engineering team) with central responsibility to define technical requirements, perform technical tradeoffs, assess risk, and evaluate evolving products?
- Is an actual rate of technical accomplishment periodically compared to a planned rate?
- Has a set of risk parameters critical to project success been delineated and periodically reviewed?
- Are estimates of cost and schedule for the completion of the project periodically assessed and updated?
- Is there a prioritized ranking of technical requirements mutually recognized by user, customer, and contractor that is periodically updated and reflected in incremental development plans?
- Are user operational scenarios included in system and acceptance testing?

External Interface Activities Factor

This factor includes activities to assure that the project interacts with and understands the needs of the larger external environment. This is especially critical for embedded software systems.

Considerations:

- Is there an on-going liaison with suppliers of other interfacing systems/elements to assure proper interfaces and allocations?
- Is early external interface testing with outside systems part of the integration plan?
- To what degree are externally provided elements validated?
- How frequently were external interfaces modified before the preliminary design review or equivalent?
- How frequently were external interfaces modified after the preliminary design review or equivalent?

User/Customer/Contractor Dialogue Factor

Each of these parties may have diverging goals with each not fully cognizant of the others' constraints. The dialogue process addresses the reconciliation of these goals to promote a win-win situation satisfactory to these constituencies. Boehm's Theory-W for software project management establishes principles for this dialogue [13].

Considerations:

- If multiple user organizations are involved, how well are users' needs being managed and reconciled?
- To what degree is there ongoing collaborative contacts between user(s), customer, and contractor to assure the correct content is in the technical requirements?
- To what degree does the user(s) participate in formal design reviews?

- Are the user(s) and contractor represented on the customer's change control board?
- Is the user-customer-contractor interaction addressing a post deployment support approach?

THE PROJECT ADVERSITY SUB-MODEL

The effect of the five adversity factors noted on Figure 1 - project size, project character, external interfaces, business constraints, and technical constraints - offset the management power factors to estimate the net turbulence of a project. A causal effect schema between the adversity variables, management power variables, and project performance is outlined momentarily. The considerations of each adversity factor are delineated below. These also have a scale of discrete responses not reproduced here.

Project Size and Character Factor

The sheer magnitude, complexity, and difficulty of the system is a reasonable first approximation of adversity.

Considerations:

- Approximately how many new lines of source code must be developed?
- How many distinct user agencies or organizations are involved?
- How many parallel operational versions of the same software for separate installations must be maintained?
- What is the degree of user interactive operations?
- How complex is the overall architecture of the system?
- If the software of this system failed, what would be the most severe impact?
- What degree of new technology (e.g. algorithms, security, protocols) development is required?
- How stringent are the real-time aspects of this system?

External Interface Adversity Factor

A major aspect of complexity and adversity is the degree that this system must interact with outside systems and elements.

Considerations:

- How many major external systems or elements does this system integrate or interoperate with?
- How many of the above interfaces require real-time or on-line synchronization?
- To what degree did externally supplied components meet technical expectations?
- How many interoperating systems or elements are undergoing development in parallel with this system?

Business Constraints Factor

A major challenge of project management is to achieve a balance between technical scope and assigned cost/schedule resources. When the cost and schedule are insufficient to meet the technical requirements, heroic and skillful management efforts become necessary to achieve even a partial success. Project managers usually understand the severity of the imbalance even at project inception, but may not be able to influence a more favorable balance because of various contractual, political, or business factors.

Considerations:

- How sufficient was the original cost baseline for this project?
- How realistic were the original key milestone dates for this project?

Technical Constraints Factor

Another source of adversity is the availability or scarcity of technical resources.

Considerations:

- How mature is the technical software engineering process used by the developing organization?
- How adequate are the available computer resources for field operation of the software?
- How adequate are the available computer resources for development of the software?
- Are the implementation standards a good fit to the size, type, complexity, and criticality of the project?

PROJECT PERFORMANCE SUB-MODEL

This sub-model embraces the business performance and technical performance factors which are shown on the left side of Figure 1. The considerations of each performance factor are delineated below. These questions also have a scale of discrete responses.

Cost and Schedule Performance Factors

Recounting both cost and schedule performance data is problematic because many projects undergo technical, cost, and schedule scope changes before they are completed. Changes to these baselines occurring late in the project may simply reflect the de-facto situation that cost/schedule and technical scope were imbalanced previously, and the change merely formalizes a reality that had been present for some time. Hence, use of final recorded cost and schedule variances could be an inaccurate gauge of business performance. Recovery of useful business performance data may necessarily rely on the subjective recollection of the survey respondents. A general guideline is that cost/schedule performance information is desired that represents the predominant baseline that existed over the longest duration of the project's development cycle. A more specific guideline is to estimate cost/schedule performance against the last project baseline change before the midpoint of the development schedule; this should discriminate "intelligent" scope changes from late reactive changes that might mask the true variances.

Cost performance factor consideration:

- To what degree did this project meet cost targets for its predominate budget and technical scope baseline?

Schedule performance factor considerations:

- To what degree were key integration milestones with interoperating systems or external elements achieved on schedule?
- To what degree did this project achieve its initial operating capability on schedule?

User Satisfaction Factor

The most obvious and visible issue of success is user acceptance of the system. This is key whether the user is external or internal to the organization that developed the software. User satisfaction may change over time as modifications are made based on initial operational experience.

Considerations:

- How satisfied were the users with system performance at initial delivery?
- How satisfied were the users with system performance six months after initial system delivery?
- How satisfied are users with system performance today?

Requirements Achievement Factor

Achievement of requirements does not necessarily guarantee user satisfaction especially if the requirements do not accurately represent the user's needs. In this investigation it is important to understand the interplay between user satisfaction, requirements achievement, and the user/customer/contractor dialogue with respect to project success.

Considerations:

- To what degree were specified functional requirements satisfied?
- To what degree were specified performance requirements satisfied?
- To what degree were specified external interface requirements satisfied?
- To what degree were specified user operational scenarios satisfied?
- To what degree were specified quality requirements (e.g., reliability, security, safety, maintainability, expandability, etc.) satisfied?

CAUSAL RELATIONSHIPS AND MODEL HYPOTHESES

The specific causal relationships between the adversity factors, management power factors, and project performance are delineated here. These relationships provide the structure for the formation of the net turbulence parameter that is hypothesized to relate significantly to project performance. During the conception of this causal model, it became necessary to envision the interaction of project adversity and management power at a more detailed level involving the individual considerations of some of these factors.

These interactions, constituting net turbulence, are diagrammed on Figure 3. The additional pseudo-factors of business risk management, technical risk management, external interface management, and user need management are introduced to detail the combination of adversity and management power parameters. The business risk management pseudo-factor is representative of this combinatorial concept. Here, business (cost/schedule) constraints were identified as a key adversity aspect that must be managed and controlled. The items that are hypothesized to overcome this type of adversity are a clear technical scope definition and the existence of a set of prioritized requirements that are incorporated into incremental development plans; these would help achieve a working partial system within the cost/schedule constraints.

The other pseudo-factors follow this same concept of identifying a specific adversity that must be controlled and hypothesizing the specific management variables that would neutralize the adversity. There is much to recommend in this line of thinking as an operational model for software project management.

The remaining management power factors of user/customer/contractor dialogue, personnel resources, physical resources, strategic planning, and project planning/control are hypothesized to have a general positive impact on project technical and business performance. Likewise, the remaining project adversity factors of project size/character and technical constraints are forecast to generally influence project performance negatively. The combined effect of these factors and the pseudo-factors is the net turbulence.

Figure 3 shows forward feeding paths to project performance labeled with "+" or "-" characters signifying the direction of contribution to performance. Also included are feedback loops that suggest the key dynamic interactions between parameters that would cause variation over project lifetime. There may be potential and value to eventually construct a system dynamics model of the broad software management process based on these causal relationships. At present, however, the compelling need is much more fundamental: to understand the degree of influence of management variables and project attributes on the technical and business performance of projects.

In summary, the basic hypotheses for exploration are stated below in natural language:

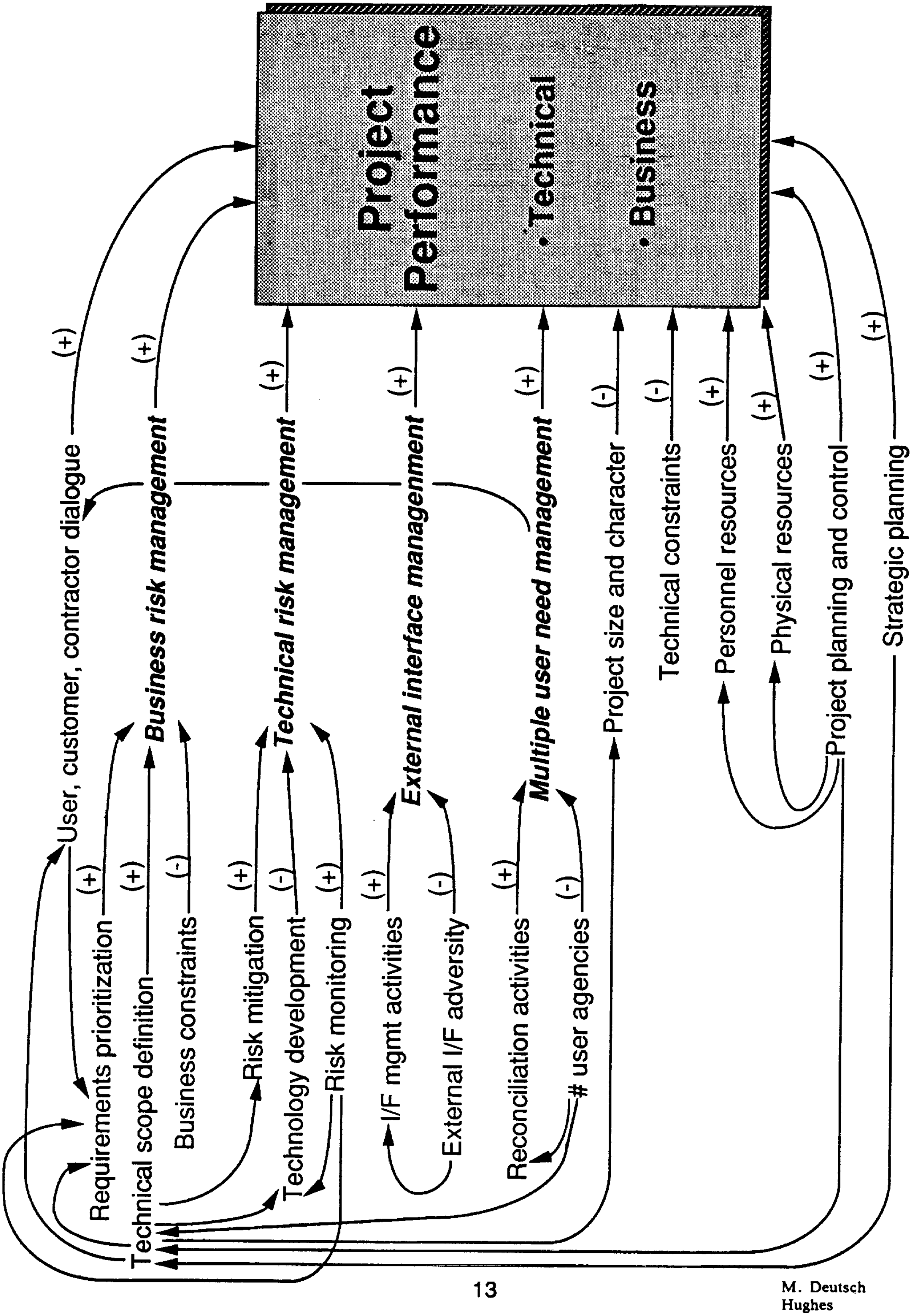


FIGURE 3: UNDERLYING CAUSAL RELATIONSHIPS

1. Degree of business risk management - consisting of the net effect of business constraints, technical scope definition, and requirements prioritization - has a significant positive effect on project performance.
2. Degree of technical risk management - consisting of the net effect of technology development, risk mitigation measures, and risk monitoring - has a significant positive effect on project performance.
3. Degree of external interface management - consisting of the net effect of external interface adversity and interface management activities - has a significant positive effect on project performance.
4. Degree of multiple user need management - consisting of the net effect of number of user agencies and multiple user reconciliation activities - has a significant positive effect on project performance.
5. The degrees of user/customer/contractor dialogue, personnel resources, physical resources, strategic planning, and tactical project planning/control each have a significant positive effect on project performance.
6. The degrees of project size/character and technical constraints each have a significant negative effect on project performance.
7. The level of net turbulence - consisting of the net effect of all the above factors - has a significant influence on project technical and business performance.

EXPLORATORY DATA ANALYSIS

An exploratory data collection and analysis on 24 projects was undertaken in order to gain insights into the feasibility of the hypothetical model diagrammed on Figure 1 and its associated hypotheses. The data collection instrument was an informal, unvalidated questionnaire with content very similar to the factors and considerations discussed in the prior sections of this paper.

The respondents to the questionnaire were an opportunistic mix of software managers and senior engineers who are developers of software systems or customers who sponsor software system developments. Some of the respondents had access to their historical files while some did not. The data is thus of varying reliability. Hence, any conclusions or insights from the analysis are solely exploratory to sharpen hypotheses for further formal study. No inferential statistical meaning is attached.

The exploratory analysis is summarized here by navigating through the relationships of the model depicted on Figures 1 and 3. The dependent measures of success are examined first followed by a probing of the relationships between the predictive and dependent measures.

Each respondent who filled out the exploratory questionnaire was asked to indicate whether the subject project was considered to be successful or unsuccessful. Figure 4 shows a plot of business performance score versus technical performance score for 24 projects with a discriminating coding for successful and unsuccessful projects as perceived by the respondents. The scoring was constructed simply by projecting the respondents answer to the discrete choices of each question on to a 10 point scale (10 is most favorable, 0 is most unfavorable), averaging the responses for each factor, and averaging the factors for the overall performance scores. The graphical plot portrays the appearance of a bimodal distribution suggesting these key issues:

- Respondents view successful projects, unsurprisingly, as combined technical and business successes (note the aggregation of successful projects in the upper right corner).
- Is there a perceived threshold of unacceptable business performance (at about business performance score "5") regardless of level of technical performance?

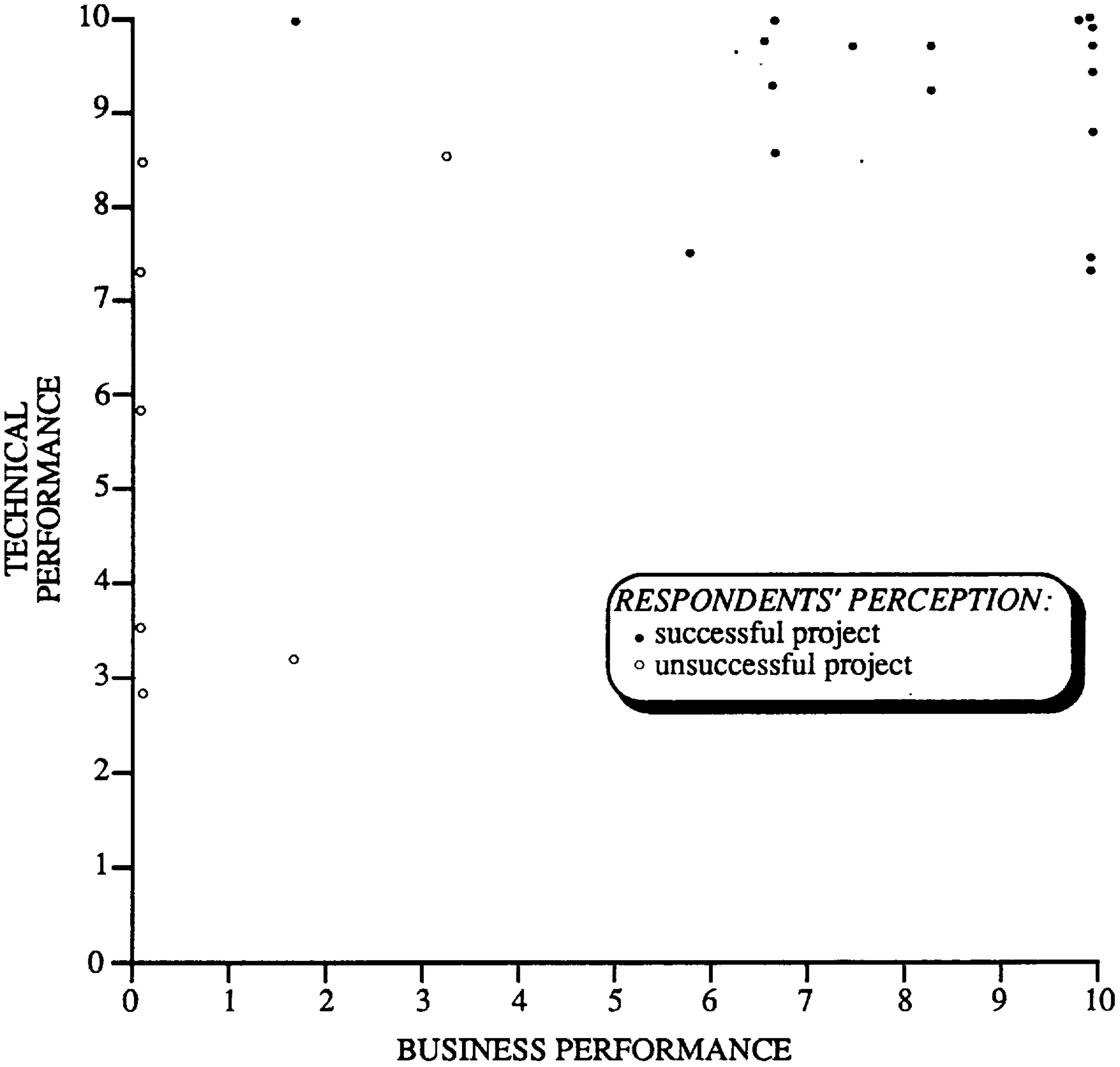


FIGURE 4: TECHNICAL/BUSINESS PERFORMANCE RELATIONSHIP

A "5" on the business performance scale corresponds to approximately a 25-50% cost overrun and a 3 month schedule slip. The existence of a perceived threshold for business performance on successful projects is identified as an additional hypothesis for further study.

The overall thesis of the hypothetical model exhibited on Figure 1 is that there should be significant correlation between the predictive measure of success, net turbulence, and the dependent measures, business and technical performance. The graphs of net turbulence against project technical and business performance, plotted on Figure 5, visually shows a general and at least moderate correlation between these variables. The Spearman non-parametric rank coefficient of correlation for technical performance on these 24 projects is 0.66 and for business performance is 0.63. This indicates that the probability of a chance correlation is less than 0.002, below the traditional 0.05 threshold. This correlation level seems even more significant, qualitatively, since the business performance (cost/schedule) component is inherently "noisy" data because of the retrieval difficulties annotated previously. Thus, the basic hypothesis that net turbulence should be significantly correlated with, i.e., predictive of, project performance is evidently feasible and should, thus, be pursued further.

TABLE 1: COMPOSITE FACTORS/PERFORMANCE RELATIONSHIP

	All projects		High adversity projects	
	Technical performance	*Business performance*	*Technical performance*	*Business performance*
Net turbulence	0.66	0.63	0.66	0.77
Technical risk management	0.43	0.25	0.65	0.59
Business risk management	0.58	0.70	0.58	0.82
External interface management	0.50	0.62	0.51	0.72
Multiple user need management	0.59	0.60	0.70	0.49
User/customer/contractor dialogue	0.48	0.49	0.56	0.44
Strategic planning	0.30	0.02	0.42	0.41
Personnel resources	0.64	0.68	0.80	0.82
Physical resources	0.25	0.44	0.34	0.62
Project planning and control	0.69	0.41	0.73	0.71
Project size and character	-0.06	-0.34	0.07	-0.11
Technical constraints	-0.48	-0.27	-0.55	-0.28

A full report on the contributions to project performance of each of the composite factors included in the causal model displayed on Figure 3 is enclosed on the matrix in Table 1. This delineates the intercorrelations (Spearman coefficient) of these factors against technical performance and business performance in two categories--all projects and only higher adversity projects of score >0.5. The following observations are made concerning the set of all projects:

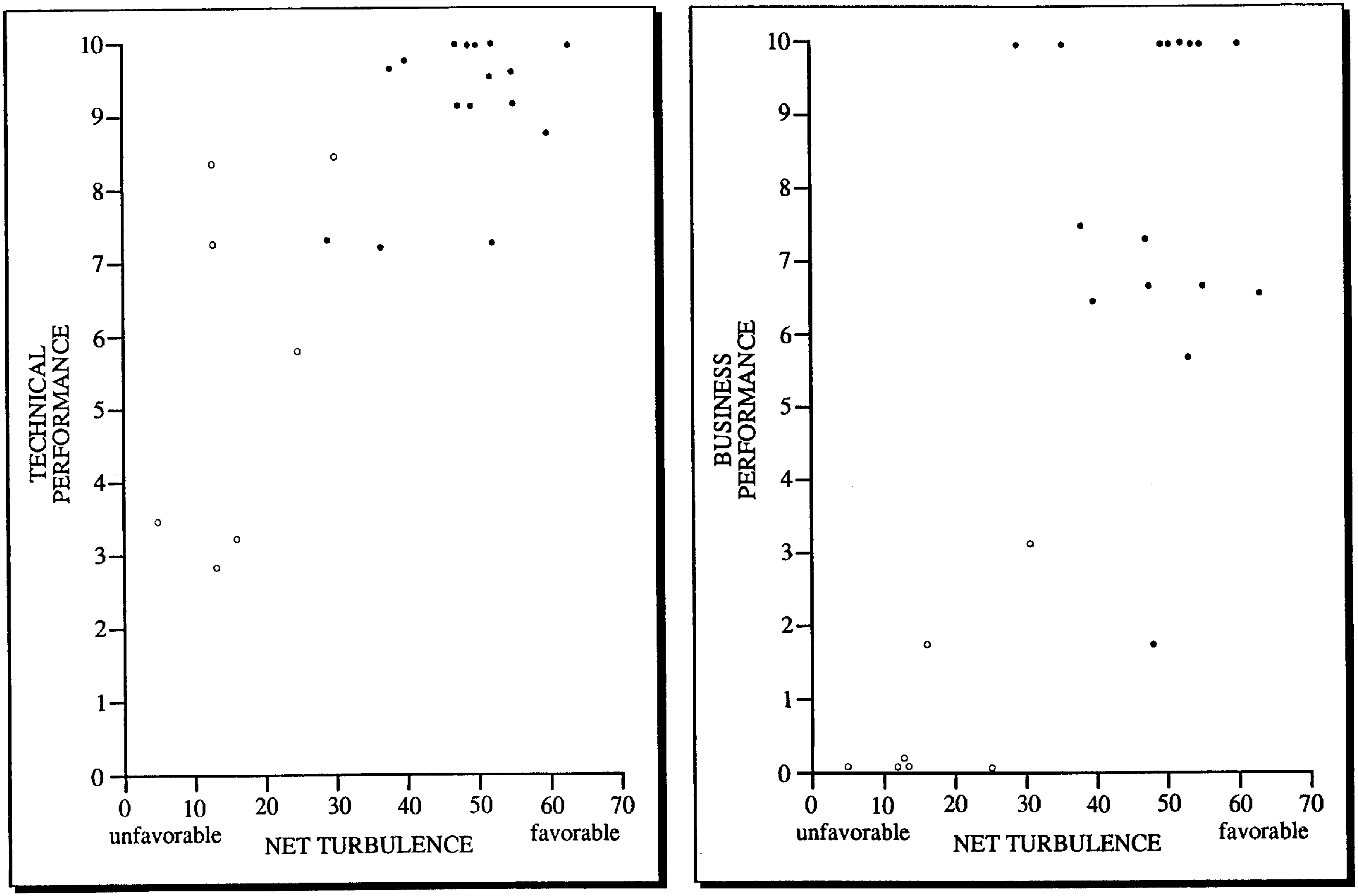


FIGURE 5: PROJECT PERFORMANCE/NET TURBULENCE RELATIONSHIP

1. The strongest contributing composite factor to performance, overall, is quality of personnel resources.
2. Other factors highly influential on performance are business risk management, external risk management, multiple user need management, and project planning/control.
3. Counter-intuitively, strategic planning and project size/character have a lesser effect on performance than expected.

Some contrasting potential tendencies for higher adversity projects are:

1. Most factors display a larger performance influence than for the full project set, especially degree of technical risk management, strategic planning, physical resources, and project planning/control.
2. Counter-intuitively, project size/character exhibits a declining effect on performance. A subset of the considerations that aggregate into this factor do show more substantial performance relationships, particularly code volume to be developed.

The above views on adverse projects suggest a further general hypothesis for consideration: management power and its factors will be more significantly correlated to project performance for higher adversity projects. This reflects the need for a more complete and sophisticated management mechanism on difficult, complex systems.

The contrast between the general set of projects and adverse projects is displayed in more detail on Table 2. The dozen individual consideration questions (components of the factors on Table 1) most correlated to technical and business performance are exhibited for each project category. The following key and interesting observations emerge for the complete project set:

1. In integrating across technical and business performance, personnel quality in three categories are significantly correlated with success - experience in functional disciplines and applications, skill of test/transition team, and skill of initial maintenance team. This illuminates the positive significance of personnel retention as the project progresses from development into operations and maintenance.
2. The existence of a central project function for technical definition and control (e.g., a system engineering organization) appears as a major driver on technical success.
3. Individual aspects of technical scope definition, user/customer/contractor dialogue, interface management, and risk control are present as significant performance contributors consistent with intuition.

The adverse projects exhibit many of the same tendencies as the full project set, but also display some significant contrasts:

1. The drivers on business performance seem to be more influential overall.
2. Review of risk parameters and inclusion of user operational scenarios in testing are greater determinants in both performance categories.

TABLE 2: MOST INFLUENTIAL INDIVIDUAL MANAGEMENT CONSIDERATIONS

Technical performance (all projects)

• How well operational personnel/support requirements specified	0.77
• Engineering and application expertise of the initial maintenance team	0.73
• Central project technical definition function	0.66
• Management/reconciliation of multiple users' needs	0.62
• Cost/schedule estimates for project completion periodically assessed	0.60
• Skill level of team that remains on project through testing and transition	0.58
• How well functional requirements are specified	0.58
• Collaborative user/customer/contractor contacts to assure correct requirements	0.57
• Actual technical accomplishment periodically compared to a planned rate	0.57
• Risk reduction measures included in life-cycle before full-scale development	0.53
• Set of risk parameters delineated and periodically reviewed	0.53
• How well computer-human interface requirements are specified	0.53

Business performance (all projects)

• Engineering and application expertise of the initial maintenance team	0.73
• Cost/schedule estimates for project completion periodically assessed	0.67
• Skill level of team that remains on project through testing and transition	0.62
• Personnel experienced in required functional disciplines & applications	0.58
• How frequently external interfaces modified after preliminary design review	0.54
• Management/reconciliation of multiple users' needs	0.52
• User(s) and contractor representation on the change control board	0.52
• How frequently external interfaces modified before preliminary design review	0.48
• Prioritized ranking of technical requirements reflected in incremental plans	0.46
• User(s) participation in formal design reviews	0.41
• Collaborative user/customer/contractor contacts to assure correct requirements	0.39
• On-going liaison with suppliers of other interfacing systems/elements	0.39

Technical performance (adverse projects)

• How well operational personnel/support requirements specified	0.79
• Set of risk parameters delineated and periodically reviewed	0.78
• Personnel experienced in required functional disciplines & applications	0.73
• Collaborative user/customer/contractor contacts to assure correct requirements	0.72
• Management/reconciliation of multiple users' needs	0.71
• Engineering and application expertise of the initial maintenance team	0.70
• Skill level of team that remains on project through testing and transition	0.68
• Central project technical definition function	0.66
• How well functional requirements are specified	0.65
• How frequently external interfaces modified after preliminary design review	0.65
• User operational scenarios included in system testing	0.61
• Cost/schedule estimates for project completion periodically assessed	0.58

Business performance (adverse projects)

• Cost/schedule estimates for project completion periodically assessed	0.85
• How well operational personnel/support requirements specified	0.76
• Prioritized ranking of technical requirements reflected in incremental plans	0.75
• How well system qualification requirements are specified	0.73
• Set of risk parameters delineated and periodically reviewed	0.71
• Personnel experienced in required functional disciplines & applications	0.70
• Skill level of team that remains on project through testing and transition	0.66
• Engineering and application expertise of the initial maintenance team	0.66
• User operational scenarios included in system testing	0.66
• Actual technical accomplishment periodically compared to a planned rate	0.61
• How well operational scenarios are specified	0.59
• How well performance characteristics are specified	0.58

3. Technical performance is more influenced by interface stability, and business performance is more affected by by ranking and incremental development of technical requirements.

The individual characteristics of the adverse projects group seem to be representative of the need for a more precise management process for these projects especially regarding risk management, external interface management, and system testing.

Another exploratory path is to examine sources of project risk and hypothesize the management power variables that could successfully manage the risks. Two major risk parameters are asserted to be:

- Unrealistically optimistic cost and schedule allocations characterized by a "business constraints" rating, and
- Degree of technology development required for a project.

In the causal model depicted on Figure 3, the business constraint rating was combined with the degree of technical scope definition and degree that requirements were prioritized and included in incremental development plans forming the business risk management pseudo-factor. An example insight into this interaction is on Figure 6 where project business constraint ratings are plotted against technical performance. It can be seen that a full range of performances are possible when cost and schedule allocations are severely constrained; a "10" is most unfavorable, indicative that a miracle was envisioned when these allocations were made. The numbers plotted next to each point reflect the precision of the technical scope definition; higher magnitudes are favorable. There is now evident a tendency for the projects with severe cost/schedule constraints (say, 7.5 or greater) to attain better performance when there is better clarity of technical scope definition. It is likely on these projects that, despite unrealistic cost and schedule allocations, management is better able to effectively apply implement-to-schedule strategies when the technical goals are well established and accomplish a reasonable level of success despite the adversity. The requirements prioritization ratings, enabling implement-to-schedule strategies, overlaid on this same graph show a similar tendency. An analogous drift is also present for business performance. The correlations of business risk management with the various performance categories are rather significant as indicated on Table 1.

An analogous insight is depicted on Figure 7 where the technology development rating is plotted against technical performance. A full spectrum of performances is seen to be present. The overlaid numbers represent the additive effects of risk mitigation activity and degree of risk monitoring. There is apparent a tendency for the better performance projects to be associated with higher combined risk management/monitoring ratings. The net effect of these three parameters is the technical risk management pseudo-factor included in the causal model on Figure 3. The correlations of this factor with the various performances was shown on Table 1 with the high adversity projects much more significantly affected.

These are but two examples of a multitude of multi-factor causal relationships that present opportunities for study.

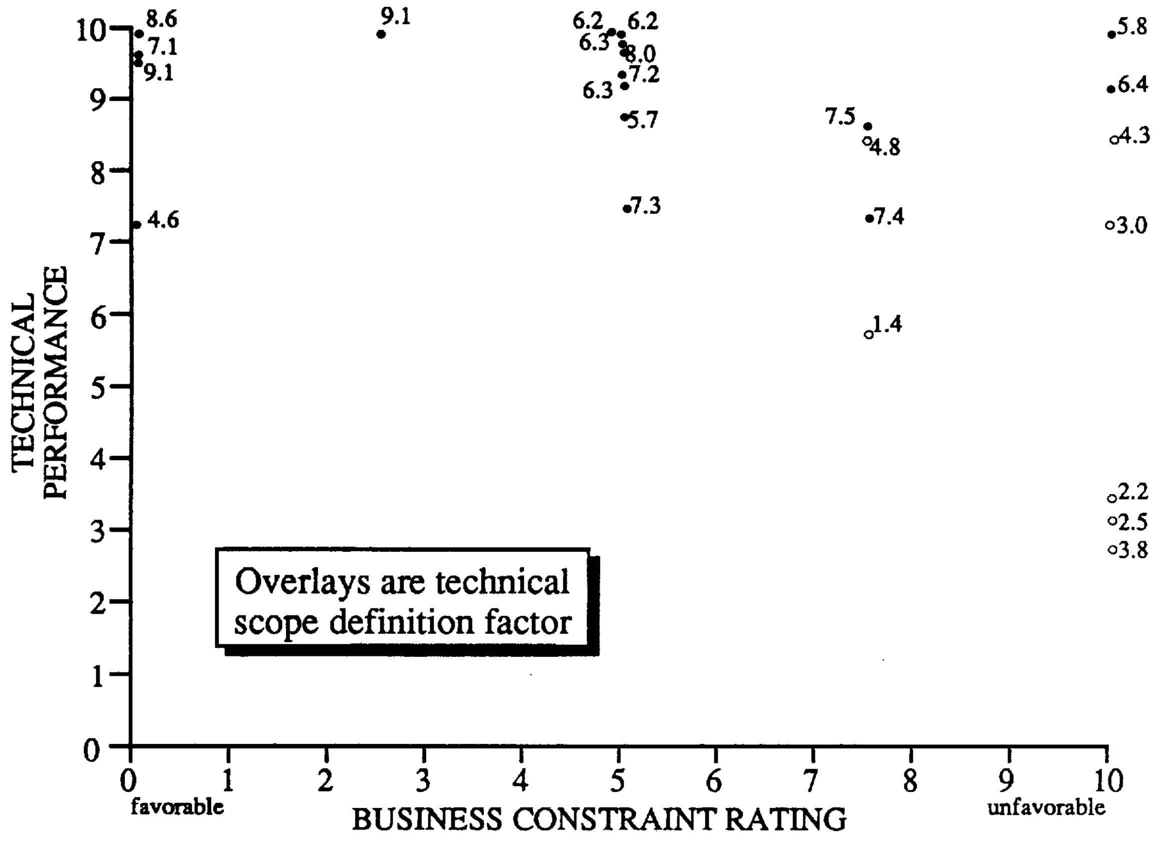


FIGURE 6: BUSINESS RISK MANAGEMENT INTERACTIONS

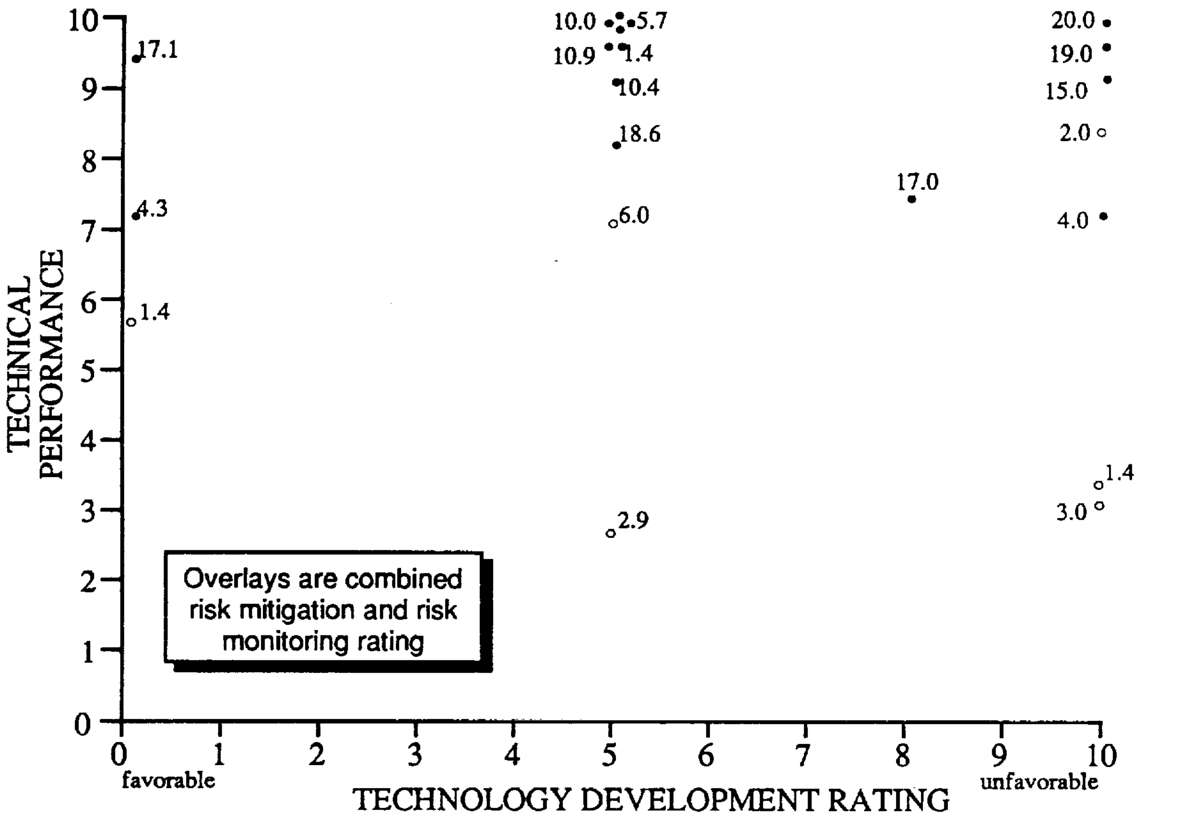


FIGURE 7: TECHNICAL RISK MANAGEMENT INTERACTIONS

CONCLUSION

The exploratory data analysis has suggested a heightened confidence in the conceptual model and its causal relationships. The key hypotheses have not been significantly contradicted with the exceptions noted. Several additional hypotheses deriving from the exploratory analysis were noted and merit further further study.

The basic eventual contributions of this study after further development and data collection are envisioned to be:

1. A general increased understanding of the dynamics and effects of project management actions;
2. An aid to practicing project managers so that they may make key management process decisions with a more visible and predictable impact on project performance; and,
3. An instructional instrument to educate project managers by systematically representing past experience in the form of lessons learned.

REFERENCES

1. D. P. Slevin and J. K. Pinto, "The Project Implementation Profile," *Project Management Journal*, September 1986, pp. 57-70.

2. J. K. Pinto and D. P. Slevin, "Critical Factors in Successful Project Implementation," *IEEE Transactions on Engineering Management*, vol. EM-34, no. 1, February 1987, pp. 22-27.

3. R. L. Schultz, D. P. Slevin, and J. K. Pinto, "Strategy and Tactics in a Process Model of Project Implementation," *Interfaces*, vol. 17, no. 3, May-June 1987, pp. 34-46.

4. J. K. Pinto and J. E. Prescott, "Variations in Critical Success Factors Over the Stages in the Project Life Cycle," *Journal of Management*, vol. 14, no. 1, pp. 5-18.

5. B. N. Baker, D. C. Murphy, and D. Fisher, "Factors Affecting Project Success," in *Project Management Handbook*, eds. D. I. Cleland and W. R. King, (New York: Van Nostrand Reinhold Co., 1983), pp. 669-685.

6. D. C. Murphy, B. N. Baker, and D. Fisher, "Determinants of Project Success," NASA NGR 22-03-028, NTIS N-74-30392, 1974.

7. W. Curtis, H. Krasner, and N. Iscoe, "A Field Study of the Software Design Process for Large Systems," *Communications of the ACM,* vol. 31, no. 11, November 1988, pp. 1268-1287.

8. A. C. Boynton and R. W. Zmud, "An Assessment of Critical Success Factors," *Sloan Management Review*, Summer 1984, pp. 17-27.

9. B. W. Boehm, *Software Engineering Economics,* (Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981).

10. R. W. Jensen, "An Improved Macrolevel Software Development Resource Estimation Model," *Proceedings Fifth Annual IPSA Conference,* St. Louis, MO, April 26-28, 1983, pp. 88-92.

11. J. Vosburgh, B. Curtis, R. Wolverton, B. Albert, H. Malec, S. Hoben, and Y. Liu, "Productivity Factors and Programming Environments," in *Proceedings of the Seventh International Conference on Software Engineering,* Orlando, FL, March 1984, pp. 143-152.

12. B. W. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer*, May 1988, pp. 61-72.

13. B. W. Boehm and R. Ross, "Theory-W Software Project Management Principles and Examples," *IEEE Transactions on Software Engineering,* vol. 15, no. 7, July 1989, pp. 902-916.

VIEWGRAPH MATERIALS

FOR THE

M. DEUTSCH PRESENTATION

Predicting Project Success from the Software Project Management Process: An Exploratory Analysis

NASA ANNUAL SOFTWARE ENGINEERING WORKSHOP
Greenbelt, Maryland
November 29, 1989

Michael S. Deutsch
Chief Scientist
HUGHES AIRCRAFT COMPANY
and
Resident Affiliate
SOFTWARE ENGINEERING INSTITUTE
Sponsored by the U.S. Department of Defense

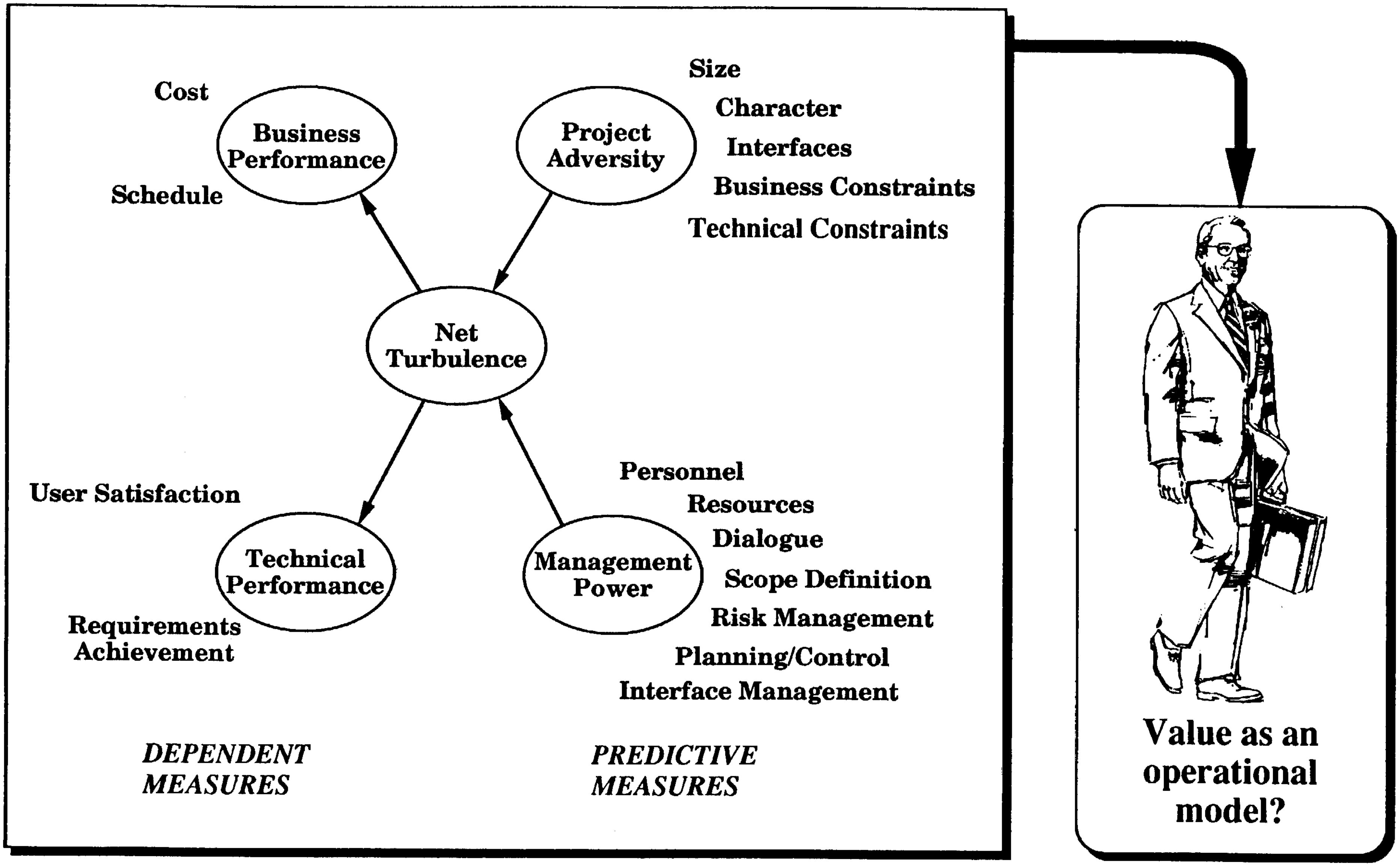
HUGHES

Empirical Project Success Study at Software Engineering Institute

- Motivation: paucity of significant empirical data on software project management process
- Goal: identify factors that discriminate between success and non-success
- Feasibility investigation--
  - ○ General understanding
  - ○ Education

Hypothetical Model of Project Success

HUGHES



Cost

Business Performance

Schedule

Project Adversity

Size

Character

Interfaces

Business Constraints

Technical Constraints

Net Turbulence

User Satisfaction

Technical Performance

Requirements Achievement

Management Power

Personnel

Resources

Dialogue

Scope Definition

Risk Management

Planning/Control

Interface Management

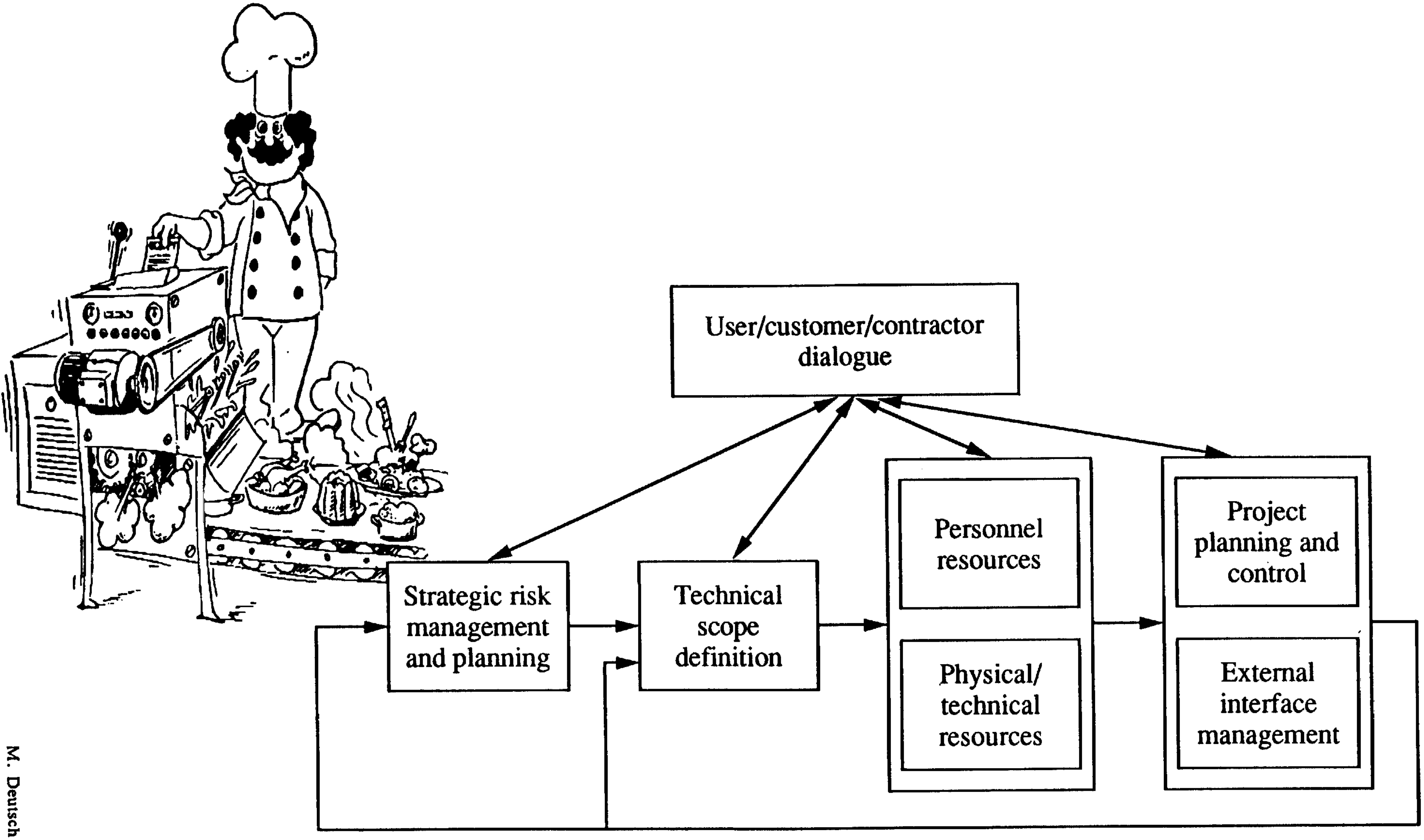
DEPENDENT MEASURES

PREDICTIVE MEASURES

Value as an operational model?

Management Process Model

HUGHES


User/customer/contractor dialogue
Strategic risk management and planning
Technical scope definition
Personnel resources
Physical/ technical resources
Project planning and control
External interface management

Example Detailed Considerations

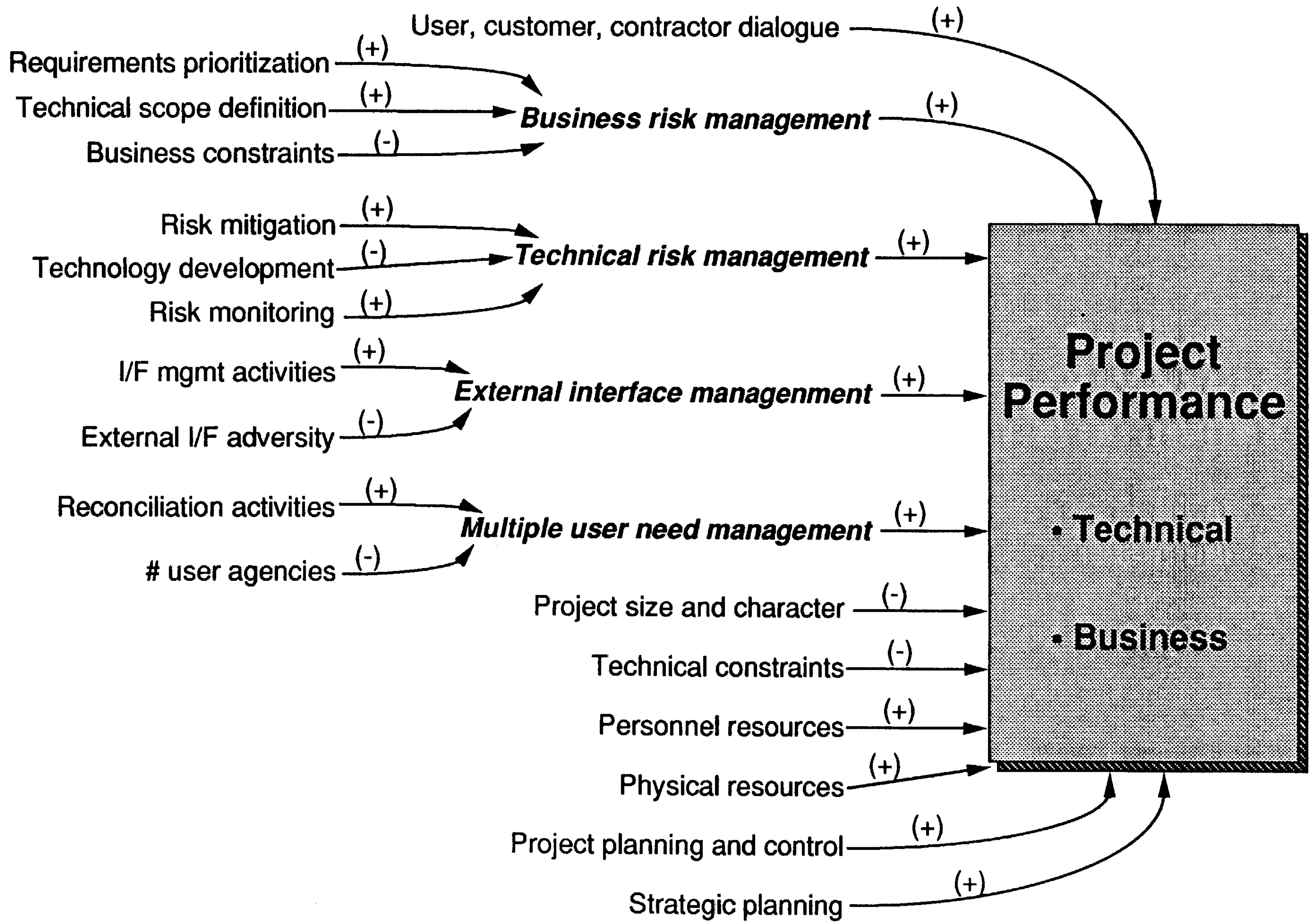
HUGHES

User/Customer/Contractor Dialogue Factor

- Reconciliation of multiple user needs
- Ongoing collaborative contacts to assure correct content in technical requirements
- User(s) participation in formal design reviews
- Representation of user(s) and contractor on customer's change control board
- Addressing of post deployment support approach

Causal Relationships

HUGHES



User, customer, contractor dialogue (+)

Requirements prioritization (+)

Technical scope definition (+)

Business constraints (-)

Business risk management (+)

Risk mitigation (+)

Technology development (-)

Risk monitoring (+)

Technical risk management (+)

I/F mgmt activities (+)

External I/F adversity (-)

External interface managenment (+)

Reconciliation activities (+)

user agencies (-)

Multiple user need management (+)

Project size and character (-)

Technical constraints (-)

Personnel resources (+)

Physical resources (+)

Project planning and control (+)

Strategic planning (+)

Project Performance

- Technical
- Business

NASA

Exploratory Data Analysis

HUGHES

Goal: examine feasibility of conceptual model

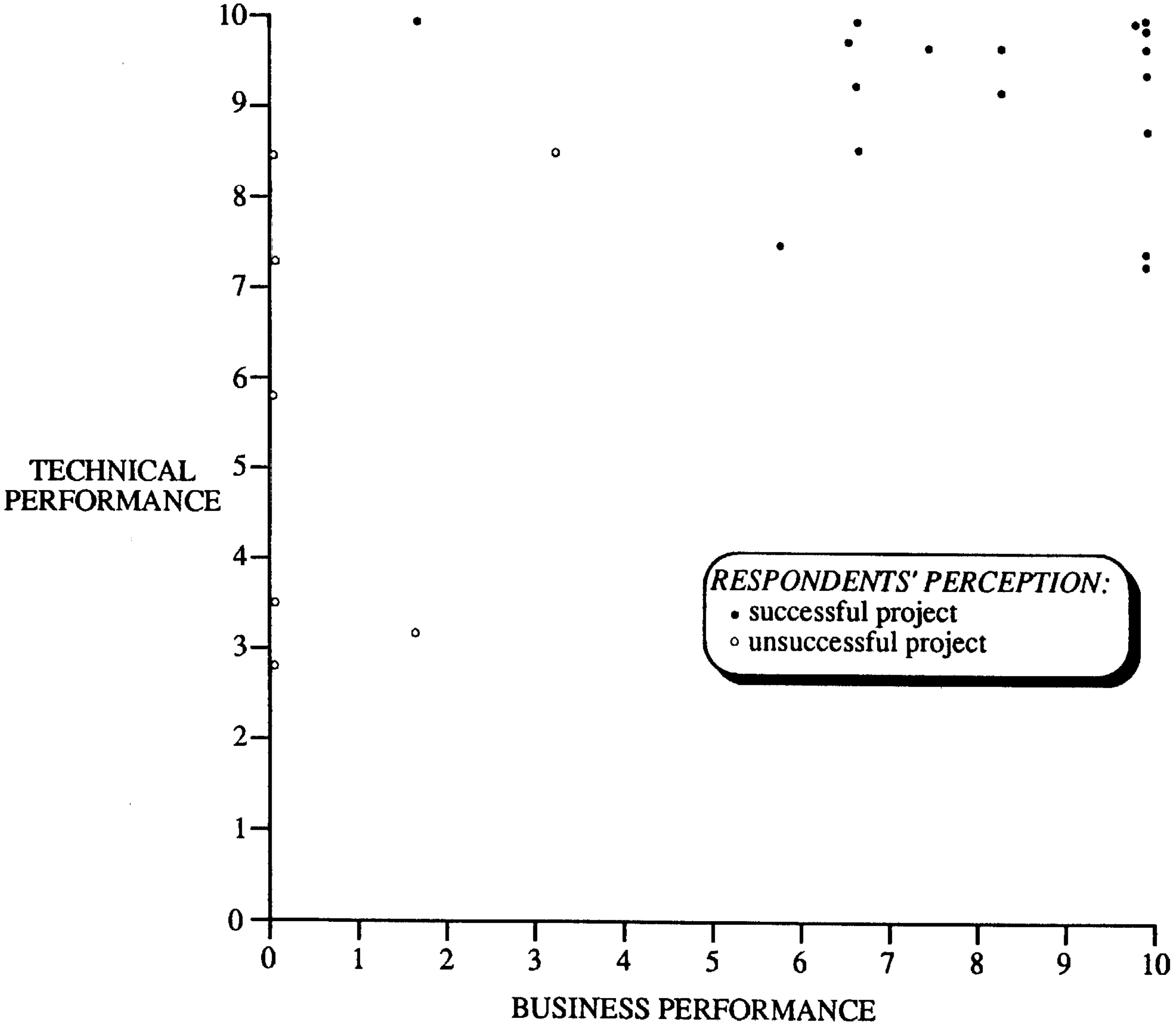
Data on 25 projects collected using informal questionnaire

Caveats on results

- Insights are pointers for future study
- No statistical inferences

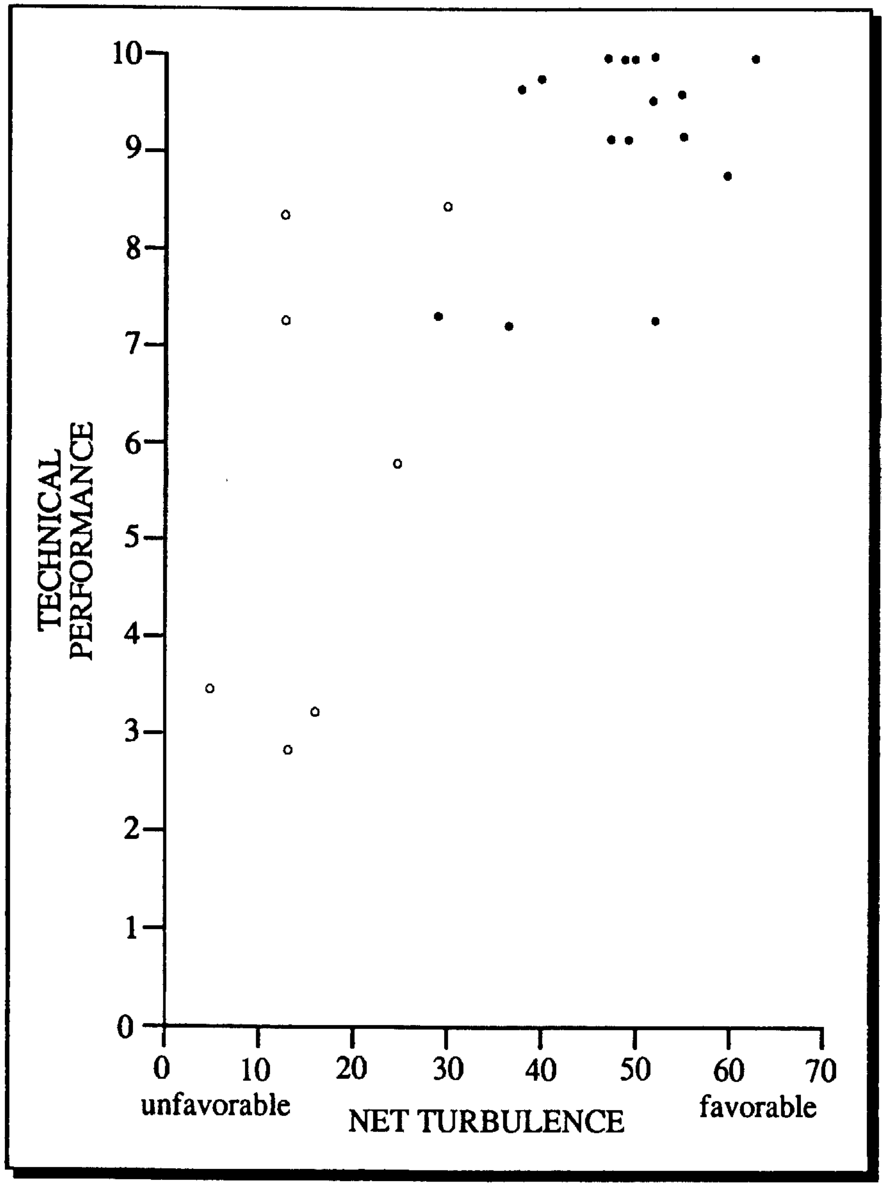
Technical and Business Performance Relationship

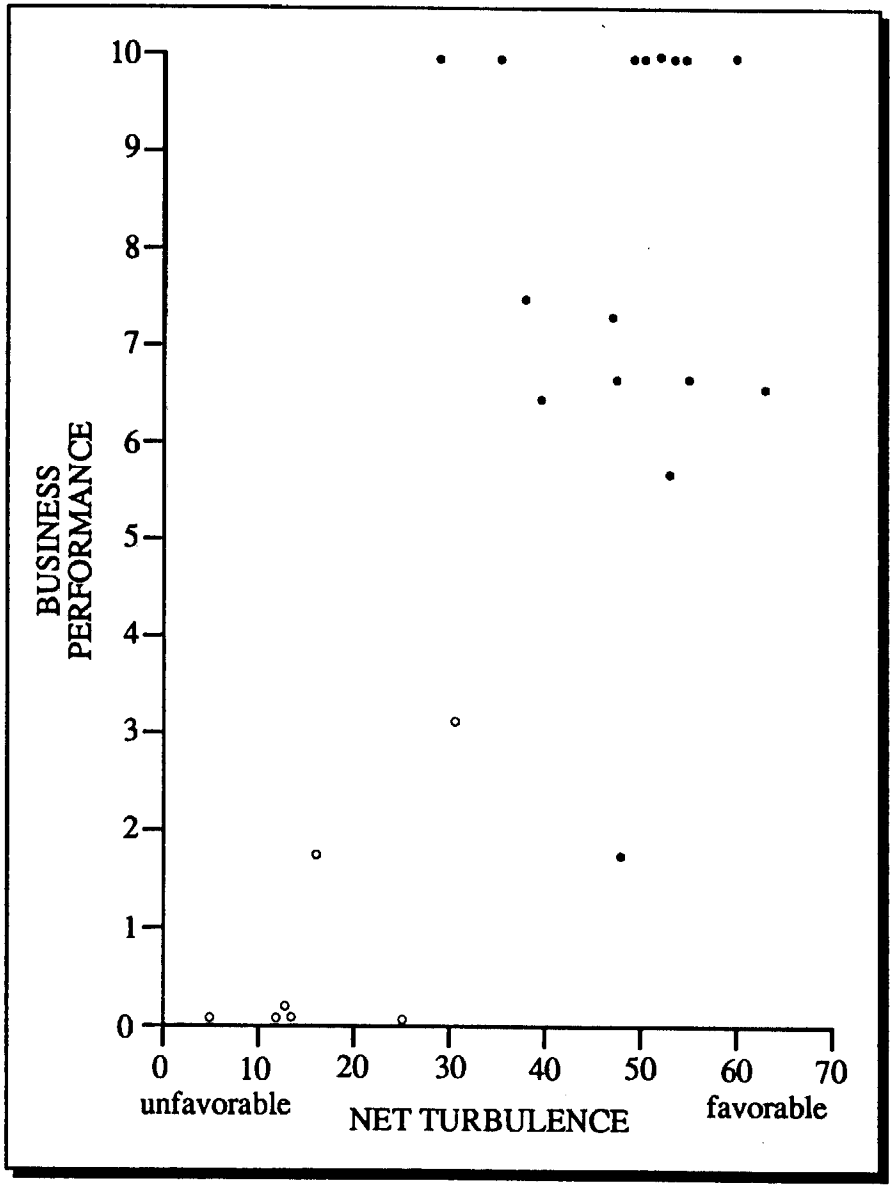
HUGHES



Performance vs Net Turbulence

HUGHES





Composite Variables & Performance

HUGHES

	All projects		High adversity projects	
	Technical performance	*Business performance*	*Technical performance*	*Business performance*
Net turbulence	0.66	0.63	0.66	0.77
Technical risk management	0.43	0.25	0.65	0.59
Business risk management	0.58	0.70	0.58	0.82
External interface management	0.50	0.62	0.51	0.72
Multiple user need management	0.59	0.60	0.70	0.49
User/customer/contractor dialogue	0.48	0.49	0.56	0.44
Strategic planning	0.30	0.02	0.42	0.41
Personnel resources	0.64	0.68	0.80	0.82
Physical resources	0.25	0.44	0.34	0.62
Project planning and control	0.69	0.41	0.73	0.71
Project size and character	-0.06	-0.34	0.07	-0.11
Technical constraints	-0.48	-0.27	-0.55	-0.28

Top Management Considerations

All Projects

HUGHES

Technical performance		*Business performance*	
• How well operational personnel/support requirements specified	0.77	• Engineering and application expertise of the initial maintenance team	0.73
• Engineering and application expertise of the initial maintenance team	0.73	• Cost/schedule estimates for project completion periodically assessed	0.67
• Central project technical definition function	0.66	• Skill level of team that remains on project through testing and transition	0.62
• Management/reconciliation of multiple users' needs	0.62	• Personnel experienced in required functional disciplines & applications	0.58
• Cost/schedule estimates for project completion periodically assessed	0.60	• How frequently external interfaces modified after preliminary design review	0.54
• Skill level of team that remains on project through testing and transition	0.58	• Management/reconciliation of multiple users' needs	0.52
• How well functional requirements are specified	0.58	• User(s) and contractor representation on the change control board	0.52
• Collaborative user/customer/contractor contacts to assure correct requirements	0.57	• How frequently external interfaces modified before preliminary design review	0.48
• Actual technical accomplishment periodically compared to a planned rate	0.57	• Prioritized ranking of technical requirements reflected in incremental plans	0.46
• Risk reduction measures included in life-cycle before full-scale development	0.53	• User(s) participation in formal design reviews	0.41
• Set of risk parameters delineated and periodically reviewed	0.53	• Collaborative user/customer/contractor contacts to assure correct requirements	0.39
• How well computer-human interface requirements are specified	0.53	• On-going liaison with suppliers of other interfacing systems/elements	0.39

Top Management Considerations

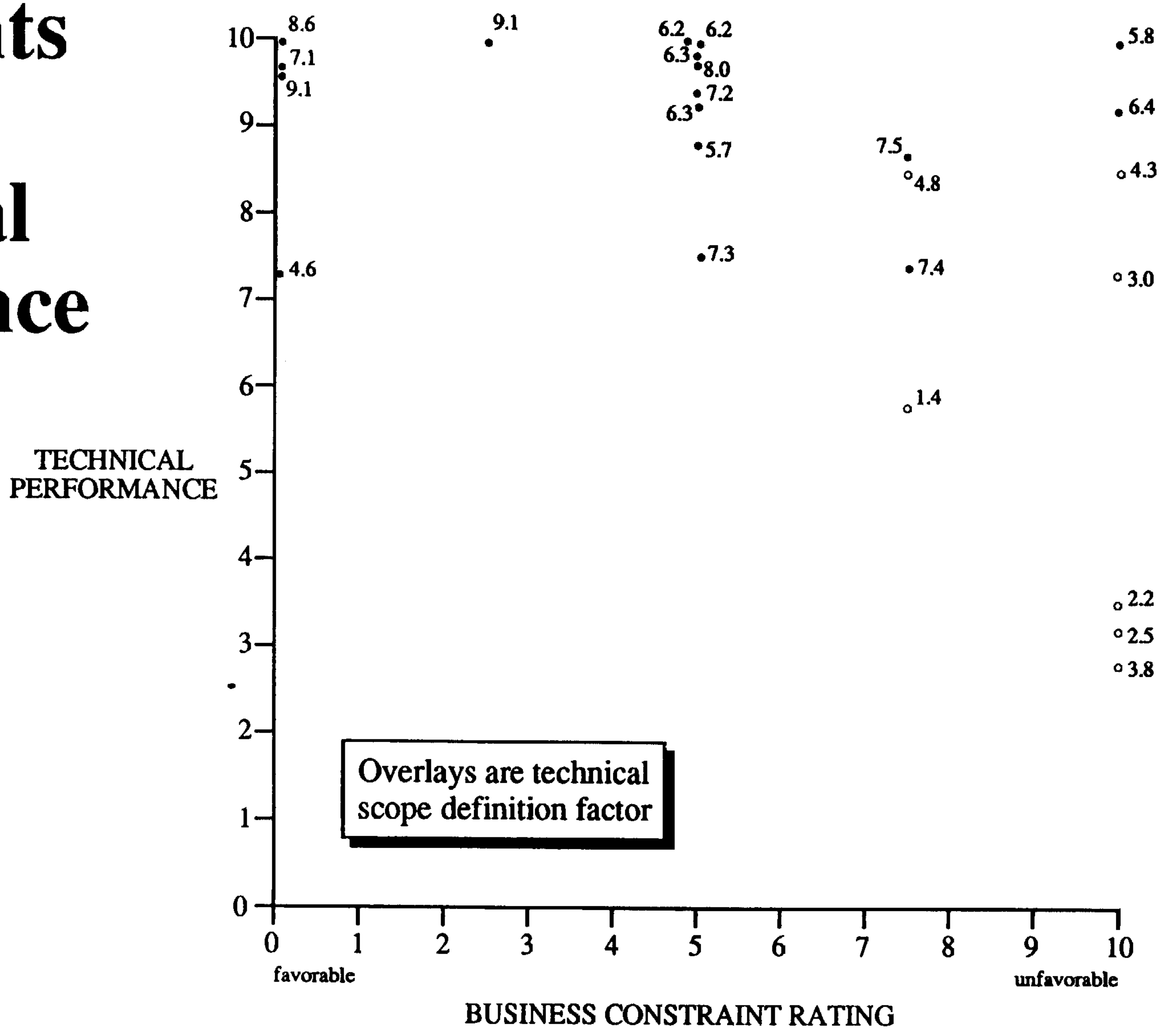
HUGHES

High Adversity Projects

Technical performance		*Business performance*	
• How well operational personnel/support requirements specified	0.79	• Cost/schedule estimates for project completion periodically assessed	0.85
• Set of risk parameters delineated and periodically reviewed	0.78	• How well operational personnel/support requirements specified	0.76
• Personnel experienced in required functional disciplines & applications	0.73	• Prioritized ranking of technical requirements reflected in incremental plans	0.75
• Collaborative user/customer/contractor contacts to assure correct requirements	0.72	• How well system qualification requirements are specified	0.73
• Management/reconciliation of multiple users' needs	0.71	• Set of risk parameters delineated and periodically reviewed	0.71
• Engineering and application expertise of the initial maintenance team	0.70	• Personnel experienced in required functional disciplines & applications	0.70
• Skill level of team that remains on project through testing and transition	0.68	• Skill level of team that remains on project through testing and transition	0.66
• Central project technical definition function	0.66	• Engineering and application expertise of the initial maintenance team	0.66
• How well functional requirements are specified	0.65	• User operational scenarios included in system testing	0.66
• How frequently external interfaces modified after preliminary design review	0.65	• Actual technical accomplishment periodically compared to a planned rate	0.61
• User operational scenarios included in system testing	0.61	• How well operational scenarios are specified	0.59
• Cost/schedule estimates for project completion periodically assessed	0.58	• How well performance characteristics are specified	0.58

Business Constraints and Technical Performance

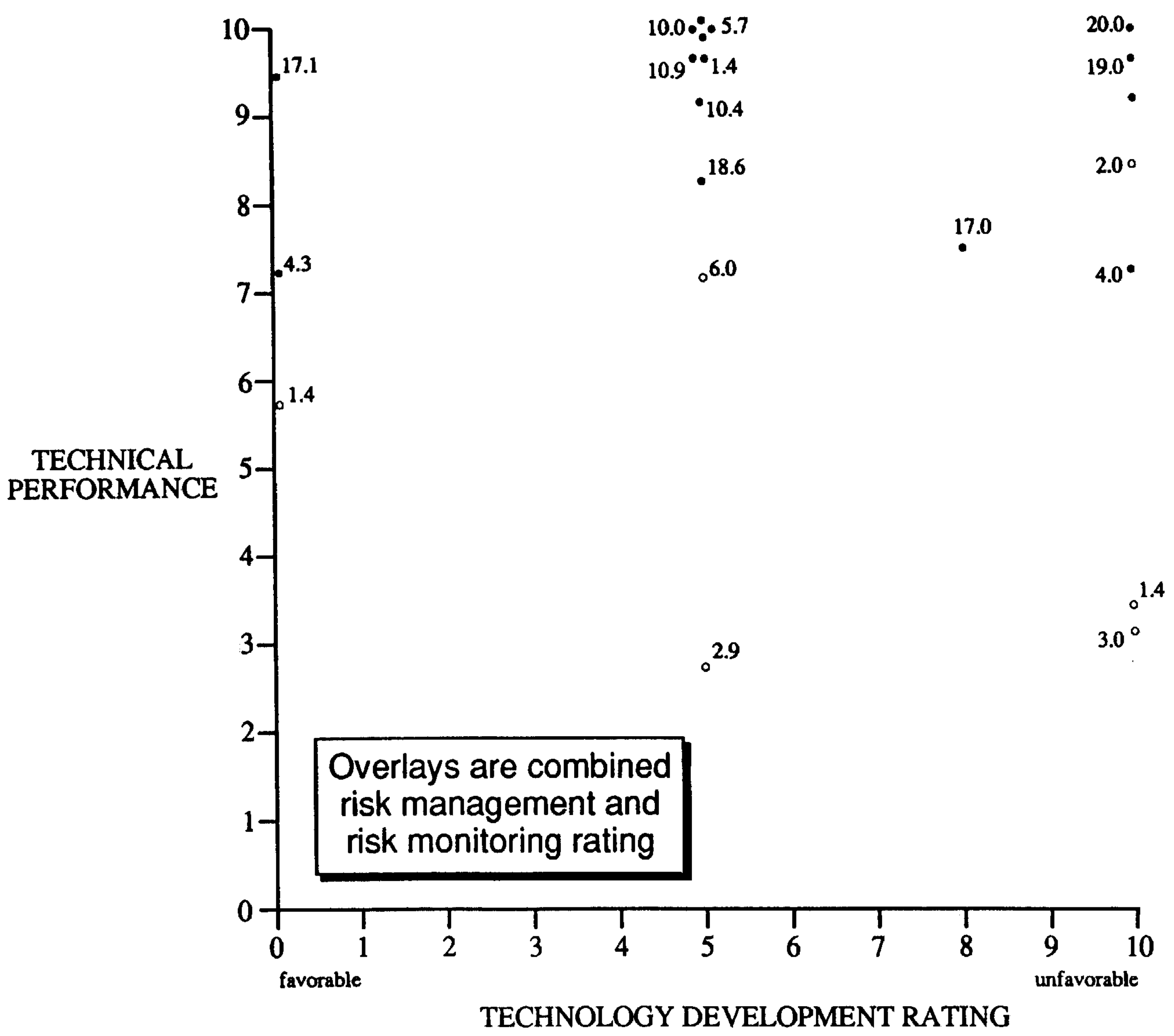
HUGHES



Overlays are technical scope definition factor

Technology vs Technical Performance

HUGHES



Overlays are combined risk management and risk monitoring rating

Key Points Summary

HUGHES

- Exploratory study provides heightened confidence in model and hypotheses
- Next steps --
  - ○ Retrospective correlational study using more rigorous methods (near term)
  - ○ Prospective applications (long term)
- Eventual contributions --
  - ○ Increased understanding
  - ○ Practical aid to program managers
  - ○ Instructional instrument

A Software Environment: Some Surprising Empirical Results

Bruce I. Blum

Johns Hopkins University/Applied Physics Laboratory
Laurel, MD 20707-6099
(301) 953-6235

A CONTEXT FOR THE ANALYSIS

A recent model of the software process describes it as a transformation from some need in the application domain to an automated product that responds to that need.[1] As shown in Figure 1, the process can be further decomposed into:

(T1) A transformation from the perceived need into a conceptual model (which uses application domain formalisms) that describes the problem and the automated solution.

(T2) A transformation from the conceptual model's view of the proposed solution into a formal specification that defines the behavior and performance of the software product.

(T3) A transformation from the formal model into an implementation.

In the traditional waterfall flow, the first two transformations are called requirements analysis and the third encompasses the software development activities.

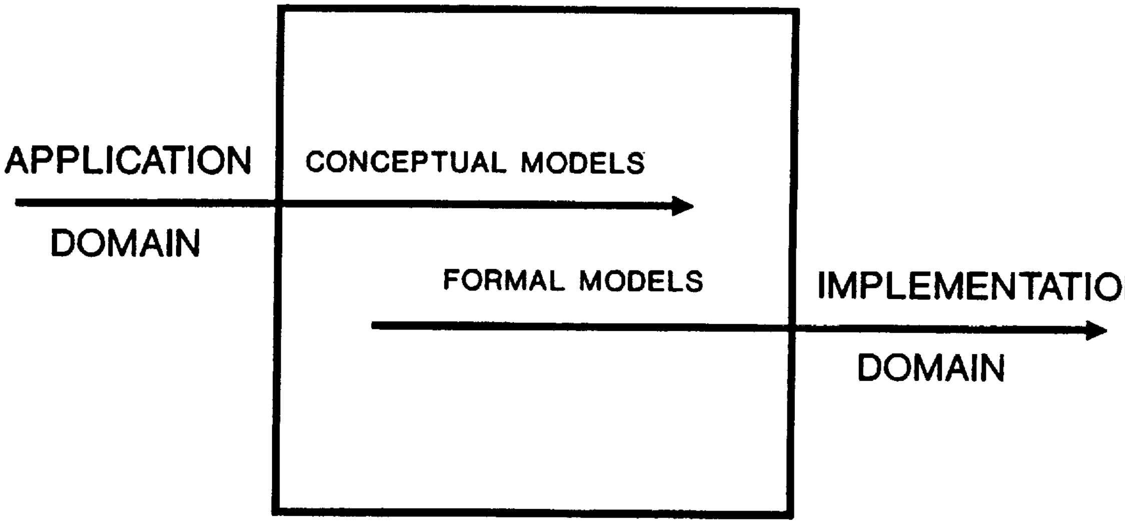


Figure 1. The essential software process.

Both T1 and T2 rely on knowledge of the application domain; they depend on the experience and judgment of the analysts. Validation increases confidence that the right system is being specified, but there is no concept of correctness. T3, however, begins with a formal specification, and there are objective criteria to determine if the product is correct with respect to its specification. Product behaviors not prescribed by the specification must be validated.

Most software problems (and associated costs) can be traced to failures in the requirements analysis phase. In terms of the software process model just presented, such failures result when the conceptual model is invalid or when the formal model is an inaccurate representation of the conceptual model. This view is reflected in Brooks' statement,

> *I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation.*[2]

(The italics are Brooks'.)

Of course, by historical necessity computer science first addressed the challenges of establishing effective representations and determining their correctness. With this orientation, the conceptual model was restricted by the technology's ability to realize implemented solutions; freedom to adapt the conceptual model was constrained by the effort to produce an implementation. Yet, today it is possible, for some well-understood domains, to bypass the problems of representation and focus only on the process of conceptual modeling (T1). This note examines how the software process behaves in such a setting.

An environment that concentrates on conceptual modeling has been used in a production setting for a decade.[3] In this environment:

> The notations used for the conceptual and formal models are isomorphic. Thus, T2 is eliminated.
>
> Behavior-preserving methods are used to transform the formal model into an implementation automatically. Consequently, T3 is eliminated.

The software process thus reduces to the implementation of prototypes that are defined, tested, and ultimately put into operational use where they continue to evolve. In Brooks' terms, the developers work only with the "conceptual construct" from which the implementation representation can be generated.

Project data from this environment provide insight into the essence of the software process when the labor of implementation is removed. As will be demonstrated in the following section, some of the results are surprising.

DATA ANALYSIS

The environment under review has been used to implement clinical information systems, itself, an AI database interface, and some smaller applications. The largest product, the Oncology Clinical Information System (OCIS), consists of over 6,000 programs and manages a database modeled with 1,700 relations; it is considered one of the most advanced systems to support tertiary care.[4] Two other hospital-based information systems have been implemented; one has been retired, and the second remains in operational use. The intelligent database interface tool is in beta test but is not used operationally. Data from these applications substantiate the following observations.

Product growth. Each product is viewed as a collection of tools by its user community. Consequently, when the system is found to be effective, new tasks are identified, and the product grows. The limits to growth are the willingness of the organization to assimilate new features, the ability of the development group to identify and implement new features, and the limits of the computational resources on which the application operates. In the case of OCIS, there is continuing growth that levels off as the equipment resources are saturated. (See Table 1.) Each new increment of computer resources is followed by an increase in system size and the number of functions supported. Growth also can be restrained. After a decision was made to replace the 750 program Core Record System, it grew by only 5% in a three year period; changes were restricted to externally-mandated modifications. In all systems studied, growth in the number of programs and number of tables (relations) in the data model is proportional. The number of new attributes in the data model grows more slowly because most new tables augment concepts associated with existing attributes.

Year measured	System size		
	Programs	Tables	Elements
1982	2,177	456	1,251
1983	3,662	848	2,025
1984	5,024	1,045	2,398
1986	5,541[1]	1,375	2,613
1988	6,605[2]	1,635	2,924

[1] Includes 399 programs not in production use.
[2] Includes 32 programs not in production use.

Table 1 Growth of OCIS at five points in its operational life.

System stability. Tables 2 and 3 illustrate the modification history of the OCIS table and program definitions. Each table presents a matrix of year initially defined by year last modified. The total column counts the number of objects defined in a year; the total row counts the number of objects last modified in that year, i.e., the object was not modified after that year. The 1988 data represent only the first 6 months of that year. The data in these tables are analyzed in the following two paragraphs.

Although the number of programs and tables grow at about the same rate, the definitions of the tables are more stable than those of the programs. For example, after OCIS had been in operational use for over 5 years, a review found that 31% of the programs and 32% of the tables were new in the sense that they had been defined in the last 2.5 years. Yet, an examination of changes to the previously-defined objects showed that 34% of the programs and only 7% of the table definitions were edited during that two-and-a-half-year period. (Reasons for the relative volatility of programs are discussed in Reference 5.)

Year defined	Year table last updated									
	1980	1981	1982	1983	1984	1985	1986	1987	1988	Total
1980	2	4	1	1	4	2	1		1	16
1981		95	31	23	5	7	11	10	3	185
1982			159	34	7	3	8	10	8	229
1983				176	19	20	13	14	1	243
1984					193	9	13	14	7	236
1985						168	15	21	5	209
1986							183	16	15	214
1987								200	20	220
1988									82	82
Totals	2	99	191	234	228	209	244	285	142	1634

Table 2 OCIS table modifications by date defined and last change.

Year defined	Year program last updated								
	1981	1982	1983	1984	1985	1986	1987	1988	Total
1980	3	5	2	5		5		2	22
1981	91	74	27	34	24	65	90	81	486
1982		335	172	52	63	151	176	199	1148
1983			252	155	102	111	162	92	874
1984				514	106	164	306	169	1259
1985					234	173	169	199	775
1986						432	306	219	957
1987							499	272	771
1988								313	313
Totals	94	414	453	760	529	1101	1708	1546	6605

Table 3 OCIS program modifications by date defined and last change.

Impact of change. In a highly integrated system one would expect changes to ripple through the system. This has been demonstrated with a small conference management example.[6] In the case of OCIS, 1,084 programs were added to the system over an 18-month period to bring its size to 6,605 programs. During this process, 2,170 previously-defined programs were edited. In other words, to increase the size of the system by 20%, some 40% of the existing programs had to be modified. While the magnitude of this change may suggest a poor design, it should be noted that the system is used to support decision making in life-threatening situations and is perceived to be free of errors. These modifications were made by a staff of 4.5 full time equivalents; many persons edited objects that they did not design.

Edits as a measure of understanding. If the software process relies on iterative problem solving, then it must involve testing potential solutions and replacing existing solutions as the problem is better understood. In this context, edits can be considered a measure of the analysts' ability to translate an idea into a valid object in the conceptual model and then evolve that object as experience accumulates. Several studies have shown that for small applications of under 100 programs an average of only 2-3 edits is required before a program is accepted as valid. (Naturally, with iterative development the needs will change, and additional edits will be required for evolution.) An 8-year edit history analysis of the 6,000 OCIS programs showed that the median number of edits for a program was 10. These edits include changes for debugging, pre-operational testing and post-operational evolution. Only 10% of the programs had been edited more than 33 times. The generation data are presented in Table 4; it is assumed that the number of edits is one less than the number of generations (compilations). Notice how stable the data are over the three periods sampled.

Percentile	Number of program generations		
	1984 study	1986 study	1988 study
10	3	3	3
20	5	5	5
30	7	7	7
40	9	9	8
50	11	11	11
60	14	14	14
70	17	17	18
80	22	23	23
90	31	31	34
99	67	82	108
Program count	4,919	5,541	6,605

Table 4 OCIS program generations for three studies.

Individual differences. There is a large body of literature suggesting that individual differences vary by as much as an order of magnitude. Although there are obvious differences among the individuals assigned to the projects examined, the range for each measure of individual performance was within 50% of the group mean. This relationship was not valid during the training period, which sometimes extended two or more years. Experience with these projects suggests that most individual differences are a training artifact rather that an inherent characteristic of the workforce. Moreover, individual effectiveness in this environment seems to correlate best with domain understanding.

Designer	Year defined								
	1980	1981	1982	1983	1984	1985	1986	1987	1988
B1	5	176	182	7					
B2			46	15					
B3		33							
B4		92	15						
B5	3	50	232	105	41				
B6		32	84						
B7	13	32							
01		19	24	40	47	1			
02			208	252	224	162	232	184	33
03	1	2	20	25	42	13	14	30	3
04		13	214	304	406	207	362	256	81
05		24	63	71	307	253	249	230	176
06			1	47	155				
07			33	5	3				
08						139	95	67	14
Other		13	26	3	34		5	4	6
Man Yrs.	–	7.0	7.0	5.5	4.5	4.5	4.5	4.5	2.3

Table 5 OCIS programs by dedigner and date of definition.

Designer	Year defined								
	1980	1981	1982	1983	1984	1985	1986	1987	1988
B1	24.8	22.6	12.6	3.6					
B2			15.4	9.9					
B3		27.0							
B4		27.5	25.2						
B5	13.0	27.7	19.0	17.5	15.8				
B6		23.3	16.5						
B7	21.6	17.9							
01		13.7	15.6	10.4	16.6	14.0			
02			23.7	15.3	15.2	13.2	10.7	9.5	9.2
03	8.0	17.5	13.3	11.7	12.5	10.7	5.6	3.2	4.3
04		33.0	18.9	12.8	15.5	20.7	10.7	7.8	8.4
05		34.2	18.9	14.0	17.1	15.9	15.4	17.2	13.6
06			16.0	21.6	18.0				
07			12.6	27.2	12.0				
08						12.3	9.5	5.9	7.8
Other		22.1	18.3	4.7	40.4		5.6	12.5	25.8

Table 6 Average OCIS program generations by designer and year of definition.

Tables 5 and 6 report annual OCIS designer activity in terms of the number of programs defined and the average number of times those programs were generated. The data are recorded for programs that were part of the production system in 1988, and there is no recording of the effort related to discarded programs. Designers from the Department of Biomedical Engineering (B1-B7) were professional software engineers, and they left the project after the system was installed. Of the Oncology Center staff, only O2 and O3 had formal training in software engineering or computer science; O3 is responsible for system management and does very little application programming. Both O4 and O5 were hired with no prior computer experience. Note the relatively high program generation rates in the first year of training; after three years on the job, however, it is difficult to identify which designers are the most experienced. (The differences in the number of programs designed by O4 and O5 in 1981-1983 relate to the fact that many of O4's initial assignments could be accomplished by copying and then editing a existing report or search program. Thus, even with a simple table like this, some domain understanding is necessary for its interpretation.)

Productivity measures. Because the environment employs a paradigm for software development and evolution limited to conceptual modeling, it is impossible to compare its productivity with paradigms that emphasize the detailing of the formal model to produce an implementation (i.e., those that focus on T3). By way of a characterizing metric, OCIS productivity over an 8-year period was .74 production programs per effort day. (See Table 5; there are approximately 225 workdays in a year.) For the prototype database interface application, the rate was 1.8 programs per effort day. These numbers do not account for discarded or retired programs. The functionality of a program is approximately the same as that of a 300-line COBOL program, but the average program length is only 15 lines. Studies of programmers in other environments suggests that a productivity rate of 15 lines per effort day is a reasonable target, so a rate of a program a day for these compact specifications should not be considered remarkable.

IMPLICATIONS

The data presented imply that the software process behaves quite differently when it is restricted to conceptual modeling (T1) without regard for program construction (T3). What has been described in this paper is the essence of building, adapting and enhancing a software product to meet a set of human needs. Today, in most application classes, we have not formalized our knowledge so that we can reuse it automatically. Thus, most of the process is concerned with what is called here the formal modeling. Because this is difficult (except where we have found ways to reuse our experience), this type of modeling activity drives the process, and conceptual modeling is restricted to expressions in the formal modeling representation scheme. I believe this distorts the software process and restricts our projects. As I have attempted to show in this paper, once the limitations of T3 are mitigated, we can recognize how dynamic and integrated our systems are. Moreover, we also can observe how effective people are at solving problems when they have access to the appropriate tools and feedback.

Of course, this does not mean that formal modeling is not important. The conceptual model referenced in this paper is a formal model. Indeed, without confidence in its formality, we would be hesitant to use the programs generated from its specifications. We label this model conceptual because it also offers a compact and expressive means for the designer to state and evaluate his or her intended goals. Naturally, the representations used by a conceptual model will be application-class specific. That is, one might expect different formalisms for interactive information systems, satellite control systems, and programming language compilers. Thus, the demonstration of a conceptual formalism in one domain may have little bearing on the utility of that formalism in some other domain. The important point, however, is that where such conceptual formalisms can be constructed, they will allow us to focus our effort on the problem we intend to solve rather than the implementation of its solution.[7] That should lead to improved productivity, reductions in apparent individual differences, the management of complexity levels otherwise considered dangerous, and (one would hope) revisions to some of the software process assumptions that we accumulated as a result of our concern for writing programs.

REFERENCES

1. B. I. Blum, Formalism and Prototyping in the Software Process, *Information and Decision Technologies*, (in press).

2. F. P. Brooks, Jr., No Silver Bullet, *Computer*, (20,4):10-19, 1987, p. 11.

3. B. I. Blum, *TEDIUM and the Software Process*, MIT Press, Cambridge, MA, 1989.

4. J. P. Enterline, R. E. Lenhard and B. I. Blum (eds), *A Clinical Information System for Oncology*, Springer-Verlag, New York, 1989.

5. B. I. Blum, Improving Software Maintenance by Learning from the Past: A Case Study, *Proceedings of the IEEE*, 77:506-606, 1989.

6. B. I. Blum, Iterative Development of Information Systems: A Case Study, *Software-Practice & Experience* 6:503-515, 1986.

7. B. I. Blum, Volume, Distance and Productivity, *Journal of Systems and Software*, 10:217-226, 1989.

ACKNOWLEDGMENT

This work was supported in part by the U. S. Navy, Space and Naval Warfare Systems Command (SPAWAR) under contract N00039-89-C-5301, task VMAR7 with the Office of Naval Research (ONR), and by the Air Force Office of Scientific Research (AFOSR) under grant AFOSR-89-0080.

VIEWGRAPH MATERIALS

FOR THE

B. I. BLUM PRESENTATION

A Software Environment
Some Surprising Empirical Results

Bruce I. Blum, Applied Physics Laboratory

The question:

How does the software process behave if we eliminate the programming activity?

The method:

Provide a conceptual model representation.

Generate the implementation from that model.

The results:

TEDIUM, an environment for developing interactive information systems (1980-)

The projects:

Oncology Clinical Information System
Core Record System
Anesthesiology System
TEDIUM
Intelligent Navigational Assistant
Small and demonstration projects

The data:

All changes and generations (who, when)
Permanent record of first and last change
Special projects

Blum, *TEDIUM and the Software Process*, MIT Press

Product Growth

OCIS

Year measured	System size		
	Programs	Tables	Elements
1982	2,177	456	1,251
1983	3,662	848	2,025
1984	5,024	1,045	2,398
1986	5,541	1,375	2,613
1988	6,605	1,635	2,924

ANES

	Programs	Tables	Elements
1981	25	6	18
1982	549	95	204
1984	1043	192	356
1986	1650	265	473

CORE

	Programs	Tables	Elements
1982	490	77	245
1984	751	136	295
1986	788	138	298

Conf. on Software Maintenance, pp. 45-56, 1987

System Stability

OCIS after 5 years of use

New objects in the 1988 OCIS
(i.e., defined in the last 2.5 years)

31% of all programs
32% of all tables (relations)

Old objects edited in same period

34% of all programs
7% of all tables

1988 OCIS 1/3 new, 1/3 edited, 1/3 stable

Impact of Change

A simple conference system

	Program group									
	1	2	3	4	5	6	7	8	9	Total
January	2	15	10							27
February		1	8			1				10
March				6	2	5				13
April					5			8	6	19
May	1				6		10	2		19
June						3			1	4
July									4	4
August					1	1		1	2	5
Total	3	16	18	6	14	10	10	11	13	101

	Program group									
	1	2	3	4	5	6	7	8	9	Total
January	6	37	16							59
February	8	8	38			7				61
March	19	5	10	9	8	23				74
April	15	4	9		20			12	13	73
May	12	12	37	2	22	5	31	2	3	126
June	6	2					1	15	15	39
July	5							1	22	28
August	5	1	3		1	2		2	9	23
Total	76	69	113	11	51	37	32	32	62	483

Software - Practice & Experience, 6:503-515, 1986.

Impact of Change (Continued)

OCIS in an 18 month period

Added 1,084 programs to the baseline

Regenerated 2,170 baseline programs

New baseline: 6,605 programs

All effort by 4.5 FTEs; a perception of no errors.

Proceedings of the IEEE, 77:596-605, 1989.

Understanding the Changes

Mean and median number of edits

Percentile	Number of program generations		
	1984 study	1986 study	1988 study
10	3	3	3
20	5	5	5
30	7	7	7
40	9	9	8
50	11	11	11
60	14	14	14
70	17	17	18
80	22	23	23
90	31	31	34
99	67	82	108
Program count	4,919	5,541	6,605

J. Systems and Software, 6:261-271, 1986.

Individual Differences I

OCIS programs by designer and year

Designer	Year defined								
	1980	1981	1982	1983	1984	1985	1986	1987	1988
B1	5	176	182	7					
B2			46	15					
B3		33							
B4		92	15						
B5	3	50	232	105	41				
B6		32	84						
B7	13	32							
01		19	24	40	47	1			
02			208	252	224	162	232	184	33
03	1	2	20	25	42	13	14	30	3
04		13	214	304	406	207	362	256	81
05		24	63	71	307	253	249	230	176
06			1	47	155				
07			33	5	3				
08						139	95	67	14
Other		13	26	3	34		5	4	6
Man Yrs.	–	7.0	7.0	5.5	4.5	4.5	4.5	4.5	2.3

TEDIUM and the Software Process, 1989.

Individual Differences II

OCIS generations by designer and year

Designer	Year defined								
	1980	1981	1982	1983	1984	1985	1986	1987	1988
B1	24.8	22.6	12.6	3.6					
B2			15.4	9.9					
B3		27.0							
B4		27.5	25.2						
B5	13.0	27.7	19.0	17.5	15.8				
B6		23.3	16.5						
B7	21.6	17.9							
01		13.7	15.6	10.4	16.6	14.0			
02			23.7	15.3	15.2	13.2	10.7	9.5	9.2
03	8.0	17.5	13.3	11.7	12.5	10.7	5.6	3.2	4.3
04		33.0	18.9	12.8	15.5	20.7	10.7	7.8	8.4
05		34.2	18.9	14.0	17.1	15.9	15.4	17.2	13.6
06			16.0	21.6	18.0				
07			12.6	27.2	12.0				
08						12.3	9.5	5.9	7.8
Other		22.1	18.3	4.7	40.4		5.6	12.5	25.8

TEDIUM and the Software Process, 1989.

Productivity Measures

An issue of granularity

If LOC/unit-of-effort is invariable over languages,
and productivity averages 2 lines/hour,
and program specification length averages 15 lines,

then we should produce programs at the rate of
one per day.

OCIS productivity (8 years):
.74 production programs per effort day.

INA productivity (12 months):
1.8 programs per effort day.

J. Systems and Software, 10:217-226, 1989.

Conclusions and Observations

System growth is constrained by mission, resources, understanding, environment, etc. We should remove the artificial constraints.

Relative to the processes, the data model is stable.

In an integrated system design, there are deep links among programs.

An environment that supports a compact and effective view of the conceptual objectives can lead to high productivity and overcome the gross individual differences.

Measurement Based Improvement of Maintenance in the SEL

H. Dieter Rombach
Bradford T. Ulery

Computer Science Department
University of Maryland

Jon Valett

NASA/GSFC

14th Annual Software Engineering Workshop
NASA/GSFC, Greenbelt, MD
November 29, 1989

1. INTRODUCTION

The SEL was created in 1977 for the purpose of investigating the effectiveness of various software engineering technologies. A large number of case studies and controlled experiments have been conducted in the past that have resulted in evolutionary changes to NASA's software development practices [*e.g.*, Basili85,McGarry85]. In this study, we have extended the traditional scope of the SEL to include software maintenance in order to gain a more complete understanding of the software lifecycle. This study began in early 1988.

This report describes some initial results of this study. The first part of the report describes the design of the study, including the goals, some background about the environment in which we are working, the improvement approach, and measurement procedures. The second part reports some of our empirical observations.

2. MAINTENANCE STUDY DESIGN

2.1. Goals of Study

The quality of the delivered product influences both what changes will be performed and the amount of effort that will be required. We therefore characterize the products with the following objectives in mind: to understand how and why the product changes; to understand how the product influences productivity during a change; and to provide historical, baseline data for future projects.

We characterize the maintenance process to understand how maintainers spend their time and what they do. We study the entire software life-cycle to understand how the maintenance process compares to development; to understand how specification developers, software developers, users, and maintainers communicate; why changes are made; and whether the organizational divisions result in the best use of personnel's skills and knowledge.

Maintenance is compared to development in order to

(1) understand the extent to which lessons learned during development can be applied to maintenance;

(2) evaluate hypotheses concerning the information loss arising from transferring a system across organizational units;

(3) evaluate the quality of the system as measured by its actual use and history of changes once development is complete.

In order to make improvements, the findings must be packaged for use by future projects. Descriptive models and baselines permit a project leader to assess a project's progress relative to past projects. Guidelines need not represent significant advances, but can serve to make projects more consistent and predictable.

2.2. Background

The focus of this study is the early maintenance phase which begins upon acceptance of the developed system and lasts until launch. During this phase, users train for launch operations and exercise the system on test scenarios.

The organizational unit responsible for maintenance consists primarily of the engineers and scientists who write the functional specifications for the software. These same people are also responsible for training postlaunch operations personnel in the use of the system. Each change, whether it is a small correction or a major functional enhancement to the system, is performed by a single person. The technology employed depends on the individual, but many of the familiar techniques from development are absent (*e.g.*, formal PDL, code reading, unit test drivers).

To date, we have monitored six projects including each of the three major types of systems developed in the SEL environment:

(1) Attitude Ground Support Systems (AGSS) provide operational support for a mission. Their functions include determining spacecraft attitude from telemetry data, verifying the on-board computer's attitude determination and control, supporting star tracking (for guidance), and more.

(2) Attitude Telemetry Data Simulator Systems produce realistic attitude telemetry and engineering data files to exercise the algorithms and processing capabilities of AGSS's. Telemetry data includes essentially everything the spacecraft knows and could report back.

(3) Attitude Dynamics Simulator Systems are analytic tools for testing and evaluating (two subsystems of) the spacecraft simulators. They simulate the environment of the spacecraft, sensor data, the on-board computer's response (actuator commands), and the resulting control torques in order to model the spacecraft dynamics.

All of the systems studied so far are written in FORTRAN (the first Ada systems are just now approaching maintenance). The systems we have studied range from 37K to 235K lines of source code (including comments and blank lines) and require from 3 to 28 staff-years to develop. The AGSS's are the larger systems, the simulators the smaller ones.

During maintenance, each change is formally defined by an Operational Software Modification Report (OSMR), a form that specifies the change, and then follows it, gathering dates and signatures as the change is approved, implemented, tested, installed, *etc.* Typically there are more outstanding OSMRs than resources. A Project Task Leader is responsible for allocating these resources.

OSMRs may be filed for several reasons. Acceptance testing may reveal the need for enhancements (corrections are still the responsibility of the software developers). Later, the users may request enhancements or identify the need for corrections or adaptations. The specification developers may also initiate changes, resulting from ideas about similar forthcoming systems. Or, the project office may modify the project requirements.

2.3. Maintenance Improvement Approach

The procedures of this study were based on the improvement paradigm (chart 4) [Basili88,Rombach88]. This paradigm suggests that maintenance can be improved by iterating the following steps for each project: (1) characterize the corporate maintenance environment; (2) state improvement goals in quantitative terms; (3) plan the appropriate maintenance and measurement procedures for the project at hand; (4) perform maintenance, measure, analyze and provide feedback; and (5) perform post mortem analysis and provide recommendations for future projects.

We apply the principles of the paradigms strictly. However, during the initial phase, our understanding of the environment, goals, and measurement procedures did not develop according to the logical ordering of the steps of the improvement paradigm [Rombach89,Rombach87]. Nor were all supporting metrics identified by a strictly top–down application of the GQM paradigm. There are two good reasons for not following these steps: (i) we sometimes discover that our knowledge of prior steps is inadequate, so we retrace our steps, or (ii) practical constraints (such as existing data collection forms) preclude a strictly top–down derivation of procedures.

Measurement procedures were validated by actually applying them to real projects on a trial basis. This trial period was an important first step in establishing the measurement program for maintenance. It gave us the opportunity to understand the environment, and demonstrate the feasibility of the planned measurement procedures.

2.4. Data Collection and Validation Procedures

We routinely monitor the effort associated with various maintenance activities, and other characteristics of the changes. Similar data is available from development. This data is used to characterize the maintenance process, the types of changes made to the product, and the reasons for making the changes [Basili84].

Routine data collection is implemented primarily through the use of forms (chart 5). At the end of each week, project personnel each complete a Weekly Maintenance Effort Form (chart 6) which briefly summarizes how they spent their time according to type of changes (correction, enhancement, adaptation, or other) and maintenance activity (isolation, implementation, unit test, integration test, other). Upon completion of each change, a Maintenance Change Report Form (chart 7) is filed. This form summarizes the change from a user's perspective (reason for change and functionality) and from the programmer's perspective (effort spent, parts of the system modified, *etc.*). A history of development (phase dates, effort) and product characteristics (size, number of subsystems, *etc.*) is available from the SEL database.

3. INITIAL STUDY RESULTS

This initial study of software maintenance within the SEL environment has concentrated on three major areas: 1) developing baselines of the software maintenance process, 2) comparing these baselines to the corresponding development baselines within the SEL, and 3) beginning to understand the problems encountered by maintenance personnel in order to provide feedback to the software developers. The following sections provide results and insights in each of these three areas.

3.1. Developing Maintenance Baselines

When attempting to measure and evaluate any software process, the SEL first attempts to establish a baseline to characterize that process. Since data collection and experimentation on the maintenance process are at an early stage in the improvement paradigm, this step is critical to the initial understanding of the maintenance process.

One way to characterize maintenance is to understand the types of maintenance requests that are being made in this environment. Maintenance requests can be broken into three categories:

Adaptations – Changing the software to conform to a new environment feature. Such items as changes due to new compilers or new operating systems fit into this category.

Enhancements – Changing the software to improve or increase its functionality.

Corrections – Changing the software to fix an error.

This data can be viewed from two perspectives, the number of changes completed and the amount of effort to perform the changes (chart 8). Determining these baselines allows for a better overall understanding of the maintenance process. These numbers do not indicate any concept of quality, they only provide a model for how this environment does its maintenance. Note that 14% of the effort during maintenance is spent performing tasks which could not be attributed to any individual change. These activities include attending meetings, management, configuration control, etc.

Interpreting this data is somewhat dangerous since its real goal is to simply define the types of maintenance requests made in this environment, however, some simple conclusions can be drawn from these baselines. Obviously, in terms of numbers the majority of maintenance changes made are error corrections, however, an overwhelming majority of the effort is spent in making enhancements to the software. This is not particularly surprising, since the enhancements might involve substantial changes to the software. One point that the data supports is that in this environment many of the maintenance requests concentrate on improving or upgrading the usability of the system. This is reflected in the amount of effort spent in making enhancements to the software. Again, this data provides a baseline of the types of maintenance requests and the effort spent in this environment. In the future this can be used as a point of comparison for new maintenance efforts.

Another characterizing model is one for predicting the cost of software maintenance. An important management tool would be the ability to determine the cost of any upcoming maintenance effort. Early research into this area has provided no insight into the "best" way to predict maintenance effort. Chart 9 shows five different models for predicting maintenance cost. For each model the a range is shown of the projects that were used in this analysis. For example, the effort per maintained million lines of code varies from 1.5 to 24 staff years, while the effort per thousands of changed lines of code varies from 0.21 to 1.25 staff years. No apparent consistent method for predicting maintenance effort can be found at this time. Given the wide disparity of the ranges, more data collection is necessary to find a strong correlation between maintenance effort and development effort or system size.

A final model of the maintenance environment that was included in this study was the change history of the product during the maintenance phase. Again, the idea is to understand how this particular environment does business, not to make value judgments. In chart 10, data on how code evolves during maintenance is presented. For each of the types of maintenance changes a bar graph showing the number of lines of code and one showing the number of components, added, changed, and deleted are shown. Regardless of the maintenance type, no or very few components are actually added to the system. This implies that the maintainers do not radically alter the system's architecture to make changes. While significant numbers of statements are added or changed during maintenance the changes do not generally involve adding components. While numerous hypotheses might be given for this lack of architecture changes, no definite interpretation currently exists.

3.2. Comparisons to the Development Environment

This study utilized the past, extensive history of the SEL software development environment as a baseline for comparison to the same organization's maintenance environment. This comparison was performed to provide insight into the similarities and differences of development and maintenance and to provide insight into how the development process affects the maintenance process. (The data used in this study to characterize the development environment consists only of those projects that were used in the maintenance study. Therefore, these results may vary slightly with those characterizing the overall development environment of the SEL.)

One area of interest in this study, was to determine the amount of effort required to isolate and repair errors in the software system. Chart 11 shows the data for isolating and completing error corrections in both the development and maintenance phases. In each table the horizontal axis represents completion effort broken down by amount of time to complete the correction, and the vertical axis represents the isolation effort by the percentage of individual changes that fall into that category. Thus, in the maintenance phase, 10% of the errors took between one hour and one day to isolate, and took longer than one day to implement or repair. One, not surprising, conclusion that can be reached from this data is that error corrections are more expensive in the maintenance phase than during development. This data simply lends support to the claim that errors introduced during design but discovered during maintenance may cost 100 times more than than if discovered and repaired during design [Boehm81]. On the other hand, another result that this data seems to suggest which is counterintuitive is that the effort to complete the corrections seems to be increasing more than the effort to isolate. Certainly, one would expect that the most difficult part of maintenance error corrections would be the isolation of the error, when in fact this data suggests that the difficulty is increased more in the actual implementation of the correction. The reasons for this trend are not clear at this time.

Another comparison which was made was to look at the types of faults that are uncovered in the maintenance and development phases. By performing this comparison, insight can be gained into the kinds of faults left in a system after acceptance testing, perhaps suggesting areas for improving software testing techniques. Chart 12 shows the percentages of faults in each of six categories for both development and maintenance. This fault data is collected in a similar manner in both the maintenance and development environment. Note that the table also shows the percentage of fault types uncovered only during acceptance testing. This was included in the study because of the possibility that the fault types changed significantly late in the life cycle and the faults uncovered during acceptance testing would therefore be similar to those uncovered during maintenance. The data implies that there are no significant differences in the types of faults uncovered during development and those uncovered during maintenance. This shows that the latent faults found in the maintenance phase are very similar to those found in the development phase. Unfortunately, this provides very little insight into ways of improving the testing or overall development process to prevent certain types of errors from occurring. These data only suggest that the types of errors uncovered in this environment are constant over the entire life cycle.

3.3. Developing for Software Maintenance

A final area of this study was to attempt to characterize software product problems encountered by maintenance personnel in order to provide feedback to software developers. To date, the data for this portion of the study has come from interviews with maintenance personnel. The early, and not surprising conclusion that can be drawn from these interviews is that the software product is not tailored to maintainer's needs (chart 13). Certainly, the suggestions of the maintenance personnel for improving the software product should be examined for future inclusion in standards and guidelines for the software development process. The primary problems maintainers are experiencing involve the actual software product.

For example, a major complaint of most maintenance personnel is that Program Design Language (PDL) is redundant and usually outdated. In the SEL environment, developers are required to keep their design PDL as part of the software module. Unfortunately, this PDL is frequently obsolete by the time the module reaches the maintenance phase, thus, it is useless to the maintainers. Also, the majority of the people maintaining the software suggested that this practice be stopped entirely, since the same level of abstraction is provided to them in the code structure and comments.

A second problem that maintenance programmers had with the software product was that global information was encoded redundantly For example, global information was encoded in multiple FORTRAN common blocks. Software modification frequently resulted in inconsistent representations of global information.

Finally, the maintainers suggested that the debug interface of the code be improved. The software developed in this environment is of a highly computational nature, therefore, in order to test the software efficiently an extensive debug interface is provided with the systems. The problem with the current debug interface is that frequently it assumes an intimate familiarity with the code in that the output was of the form <variable> = <value>. Maintenance personnel suggested that in the future debug interfaces provide a more descriptive explanation of the output printed.

4. FUTURE DIRECTIONS

We are only beginning to understand the SEL maintenance environment. Regular monitoring of all SEL maintenance projects will eventually result in better, more reliable, models.

Eventually, we will define guidelines for maintenance and development. Maintenance guidelines (*e.g.*, baseline data, explicit maintenance process models, and quality models) will provide a better basis for controlling ongoing maintenance projects, and planning future ones. Development guidelines will describe how to build software with maintenance in mind in the first place. Such guidelines assume the existence of an maintenance artifact model which captures our understanding of what information should be passed to maintainers.

As the software technology used in the SEL changes, our maintenance measurement approach needs to be refocused. Currently, we are in the process of preparing for the monitoring of our first Ada products. We are especially interested in understanding the differences between maintaining Ada versus FORTRAN software [Katz86], and whether these differences can be attributed to the Ada language itself or the supporting technology (*e.g.*, object–oriented design).

As both maintainability and reusability of a product seem to be highly dependent on its understandability and modifiability, we expect to coordinate this project more with ongoing reuse–oriented research projects in the future.

Overall, this project extends the use of measurement in the SEL to the entire software life–cycle. It should further improve our understanding, and the ability to plan and control future software developments in the SEL.

5. REFERENCES

Basili84	V. R. Basili and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," *IEEE Transactions on Software Engineering* SE–10(6), pp.728–738 (November 1984).
Basili85	V. R. Basili, "Can We Measure Software Technology: Lessons Learned from Eight Years of Trying," *Proceedings Tenth Annual Software Engineering Workshop*, NASA Goddard Space Flight Center (December 1985).
Basili88	V. R. Basili and H. D. Rombach, "The TAME Project: Towards Improvement–Oriented Software Environments," *IEEE Transactions on Software Engineering* **SE–14**(6), pp.758–773 (June 1988).
Boehm81	B. Boehm, *Software Engineering Economics*, Prentice–Hall, Englewood Cliffs, New Jersey (1981).
Katz86	E. E. Katz, H. D. Rombach, and V. R. Basili, "Structure and Maintainability of Ada Programs: Can We Measure the Differences?," *Proceedings 9th Minnowbrook Workshop on Software Performance Evaluation* (August 1986).
McGarry85	F. E. McGarry, "Recent SEL Studies," *Proceedings Tenth Annual Software Engineering Workshop*, NASA Goddard Space Flight Center (December 1985).
Rombach87	H. D. Rombach and V. R. Basili, "A Quantitative Assessment of Software Maintenance: An Industrial Case Study," *Proceedings Conference on Software Maintenance-1987*, pp.134–144 (September 21–24, 1987).
Rombach88	H. D. Rombach and B. T. Ulery, "Improving Software Maintenance through Measurement," Technical Report CS–TR–2131, Dept. of Computer Science, University of Maryland, College Park, Maryland (October 1988). Published as an invited paper, IEEE Proceedings, April 1989.
Rombach89	H. D. Rombach and B. T. Ulery, "Establishing a Measurement–Based Maintenance Improvement Program: Lessons Learned in the SEL," *Proceedings of the Conference on Software Maintenance* (October, 1989).

VIEWGRAPH MATERIALS

FOR THE

H. D. ROMBACH PRESENTATION

Measurement Based Improvement of Maintenance in the SEL

H. Dieter Rombach
Bradford T. Ulery

Computer Science Department
University of Maryland

Jon Valett

NASA/GSFC

14th Annual Software Engineering Workshop
NASA/GSFC, Greenbelt, MD
November 29, 1989

CHART 1

GOALS OF STUDY

- Characterize the maintenance process and product.

- Compare maintenance and development characteristics.

- Package findings to improve maintenance and development
(baselines, models, guidelines).

CHART 2

BACKGROUND

- **Process studied**
    - Early maintenance phase
    - Beginning early 1988
- **Maintenance organization**
    - Separate from development
    - Individual process (*i.e.*, each change is performed by one person)
    - Technology level is different from development
- **Systems studied**
    - Six attitude software systems for satellites
    - All developed according to standard SEL methodology
    - FORTRAN
    - Size: 37K to 235K lines of code
    - Development effort: 3 to 28 staff–years
- **Maintenance Tasks**
    - 146 maintenance change requests (OSMRs)
    - Request sources: maintainers, users

IMPROVEMENT PARADIGM

- **Characterize the corporate maintenance environment**
- **State improvement goals**
  - ○ State improvement goals informally
  - ○ Specify related measurement goals
- **Plan maintenance**
  - ○ Plan appropriate maintenance process
  - ○ Plan appropriate measurement process
- **Perform maintenance**
  - ○ Perform maintenance process
  - ○ Perform measurement process
  - ○ Analyze collected data and provide immediate feedback
- **Perform post–mortem analysis and provide recommendations for future projects**
- **Return to step I1**

DATA COLLECTION

- **Standard SEL Data Collection Tailored to Maintenance:**
  - ○ Weekly effort data (form):

    By class (adaptation, correction, enhancement)

    By activity (*e.g.*, isolation, change design, implement, unit/system test, acceptance test)
  - ○ Change data (form):

    Class of change

    Effort, by isolation & completion

    Degree of reuse (additions, changes, deletions)

    Source of problem (requirements, specification, design, code)
  - ○ Baseline data (interviews):

    Initial models of maintenance

    Improvement goals

    Maintenance problems (*e.g.*, tools, documentation, structure)

CHART 5

WEEKLY MAINTENANCE EFFORT FORM

Name: ______________________ Friday Date: ______________

Project: ______________________

Section A – Total Hours Spent on Maintenance (Includes time spent on all maintenance activities for the project excluding writing specification modifications) □

Section B – Hours By Class of Maintenance (Total of hours in Section B should equal total hours in Section A)

Class	Definition	Hours
Correction	Hours spent on all maintenance associated with a system failure.	
Enhancement	Hours spent on all maintenance associated with modifying the system due to a requirements change. Includes adding, deleting, or modifying system features as a result of a requirements change.	
Adaptation	Hours spent on all maintenance associated with modifying a system to adapt to a change in hardware, system software, or environmental characteristics.	
Other	Other hours spent on the project (related to maintenance) not covered above. Includes management, meetings, etc.	

Section C – Hours By Maintenance Activity (Total of hours in Section C should equal total hours in Section A)

Activity	Activity Definitions	Hours
Isolation	Hours spent understanding the failure or request for enhancement or adaptation.	
Change Design	Hours spent actually redesigning the system based on an understanding of the necessary change.	
Implementation	Hours spent changing the system to complete the necessary change. This includes changing not only the code, but the associated documentation.	
Unit Test	Hours spent testing the changed or added components. Includes designing tests and writing test drivers.	
Integration Test	Hours spent testing the components integrated into the system. Includes hours spent on system test.	
Other	Other hours spent on the project (related to maintenance) not covered above. Includes management, meetings, etc.	

CHART 6

MAINTENANCE CHANGE REPORT FORM

Name: ______________________ **OSMR Number:** ______________

Project: ______________________ **Date:** ______________

SECTION A: Change Request Information

Functional Description of Change: ____

____

____

____

What was the type of modification?

- ___ Correction
- ___ Enhancement
- ___ Adaptation

What caused the change?

- ___ Requirements/specifications
- ___ Software design
- ___ Code
- ___ Previous change
- ___ Other

SECTION B: Change Implementation Information

Components Changed/Added/Deleted: ____

____

____

	< 1hr	1 hr to 1 day	1 day to 1 week	1 week to 1 month	> 1 month
Estimate the effort spent isolating/determining the change:					
Estimate the effort to design, implement, and test the change:					

Check all changed objects:

- ___ Requirements/Specifications Document
- ___ Design Document
- ___ Code
- ___ System Description
- ___ User's Guide
- ___ Other

If code changed, characterize the change (check most applicable)

- ___ Initialization
- ___ Logic/control structure (e.g., changed flow of control)
- ___ Interface (internal) (module to module communication)
- ___ Interface (external) (module to external communication)
- ___ Data (value or structure) (e.g., variable or value changed)
- ___ Computational (e.g., change of math expression)
- ___ Other (none of the above apply)

Estimate the number of lines of code (including comments): ___ added ___ changed ___ deleted

Enter the number of components: ___ added ___ changed ___ deleted

Enter the number of the added components that are ___ totally new ___ totally reused ___ reused with modifications

5150G(1)-3

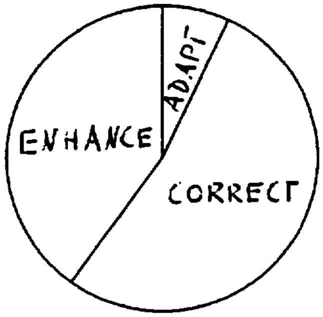
CHART 7

CHARACTERIZE PROCESS

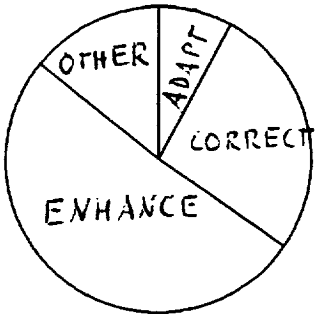
- **QUESTION:**

  What types of maintenance requests are made?

- **OBSERVATIONS (forms):**



Number of Changes



Effort

- **INTERPRETATION:**

  Enhancements may be due to

  Maintenance characteristics (*e.g.*, emphasis on improving usability)

  Development characteristics (*e.g.*, imprecise requirements)

CHARACTERIZE COST

- **QUESTION:**

  How can the cost of maintenance be estimated?

- **OBSERVATIONS:**

○ Effort, total (staff–years):	[0.07, 1.7]
○ Effort per year (% of devt effort):	[1%, 23%]
○ Effort per maintained MLOC (staff–years):	[1.5, 24]
○ Effort per "modified" KLOC (staff–years):	[0.21, 1.25]
○ Effort per 100 changes (staff–years):	[1, 15]

- **INTERPRETATION**

  No obvious correlation between maintenance effort and (development effort, system size)

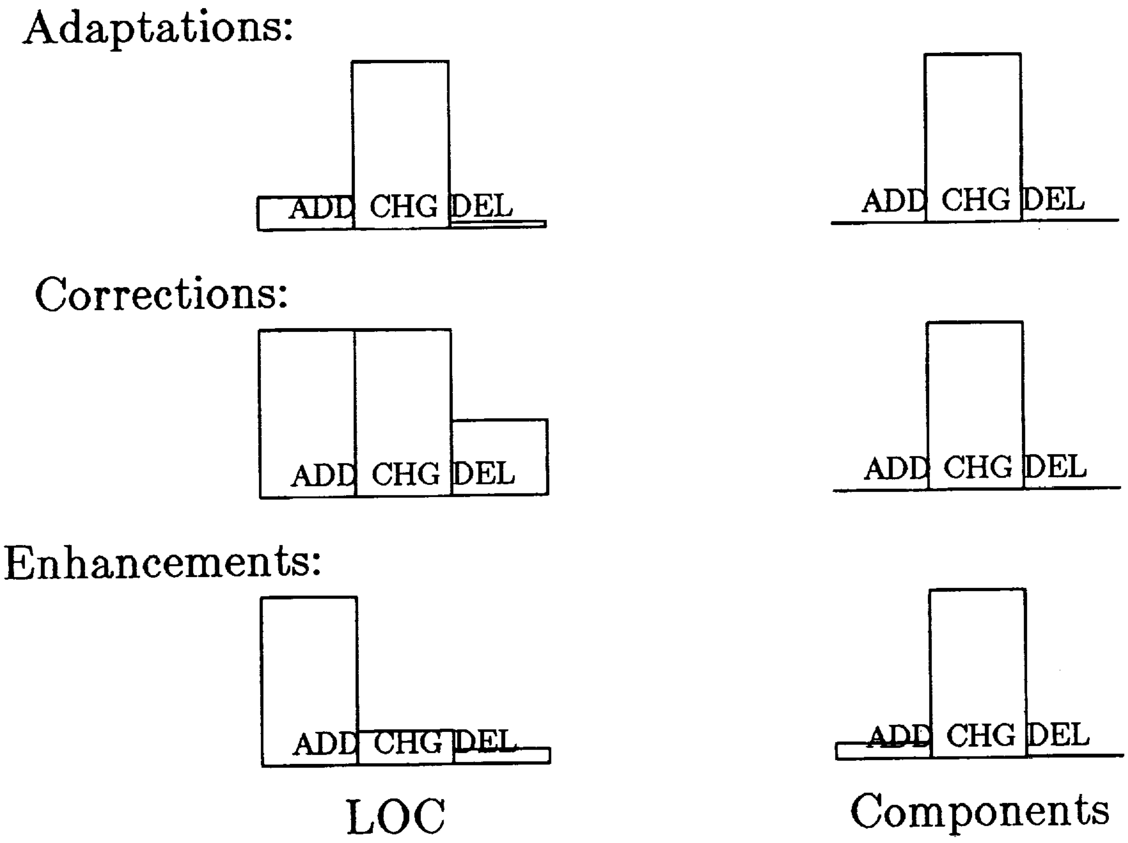
  Mainly a function of the amount of maintenance performed

CHARACTERIZE CODE EVOLUTION

- **QUESTION:**

  How does the code evolve during maintenance?

- **OBSERVATIONS (forms):**



- **INTERPRETATION:**

  Maintainers do not change the system architecture.

CHARACTERIZE CORRECTIONS

- **QUESTION:**

  How much effort is required to make corrections during maintenance and development.

- **OBSERVATIONS (forms):**

Development:

Isolation	Completion Effort		
Effort	< Hour	< Day	Longer
< Hour	48	10	1
< Day	11	16	4
Longer	2	3	4

Maintenance:

< Hour	4	16	9
< Day	0	31	10
Longer	0	9	21

- **INTERPRETATION:**

  Error corrections are more expensive during maintenance than during development.

  Effort to complete corrections increases more than effort to isolate.

CHART 11

CHARACTERIZE FAULTS

- **QUESTION:**

  Are the types of faults detected during maintenance different than those detected during development?

- **OBSERVATIONS (forms):**

  Types of faults:

Structure	Devt	Acc Test	Maint
Computation	14%	20%	11%
Data Value	25%	26%	28%
Initialization	16%	16%	25%
Ext. Interface	8%	7%	8%
Int. Interface	19%	12%	15%
Logic, Control	18%	18%	13%
Other	–	–	2%

- **INTERPRETATION:**

  The same types of faults are found in maintenance and development.

CHARACTERIZE PRODUCT PROBLEMS

- **QUESTION:**

  What product characteristics create maintenance problems?

- **OBSERVATIONS (interviews):**

  ○ Outdated PDL frustrates maintainers.

  ○ PDL is redundant; it provides the same level of abstraction as code structure and comments.

  ○ Global information encoded redundantly.

  ○ Debug interface assumes intimate familiarity with code. (<variable> = <value>)

- **INTERPRETATION:**

  The product is not tailored to maintenance needs.

FUTURE DIRECTIONS

- **Continue to build maintenance models & baselines.**

- **Provide guidelines for maintenance.**

  Baseline data

  Maintenance process model

  Quality models (*e.g.*, for resource estimation)

- **Provide guidelines to development.**

  Maintenance product model

  Development process model to support maintenance

- **Expand to monitoring the maintenance of Ada systems.**

  Model Ada products and process.

  Compare to FORTRAN observations.

  Assess implications of OOD & Ada on maintenance.

- **Learn from this study for reuse of experience in general.**

CHART 14

SESSION 3 — SOFTWARE REUSE

M. Lehman, Imperial College

J. C. Knight, University of Virginia

C. Braun, Contel

K. Thackrey, Planning Analysis Corporation

Software, Systems and Application Uncertainty and its Control Through the Engineering of Software

M M Lehman

Lehman Software Technology Associates Ltd
and
Department of Computing
Imperial College of Science and Technology

London SW7 2BZ

Abstract

Computers are being applied more and more broadly to address applications in all areas of human activity, penetrating ever deeper into the very fabric of society. As a consequence, mankind is becoming, collectively and individually, ever more dependent on **software** and on the **integrity** of that software. In this context the term software includes both the systems software that constitutes a fundamental part of the operational configuration and the programs that implement each individual application. Integrity is a many faceted concept that has to do with the **availability** of programs whenever they are needed and their **correctness** in relation to the circumstances at the **moment of execution** or, more precisely, when the results of computation are **applied**. A program must produce a solution that is **correct** and **relevant** when **used**. It must continue to do so whenever **required** over the lifetime of an application and of the systems that realise and support it. All this despite continuing change in a dynamic world.

This paper opens with a brief discussion of the fundamental concepts of software engineering. The discussion leads to the formulation of a **Principle of Uncertainty** that applies, in general, to all computer application in the real world. The principle follows because any program is a **model**, albeit many times removed by abstraction and reification from the real world it reflects and addresses. The consequences of this basic fact leads to recognition of a need for a **disciplined technology** associated with a controlled **process** for definition of each application, its operational domain and the envisaged system with its software; and for their development, application and evolution (maintenance).

The paper continues with a brief analysis of the control of uncertainty through the application of software engineering technology. This is seen as the discipline that permits one to limit uncertainty and its consequences through the introduction and control of appropriate development **processes** and the systematic and disciplined application of **methods** and **tools**. Finally the paper places the views presented into the context of other observations about the state of the art in software development and remarks on the relevance of the issues raised to society as a whole.

Keywords

Software dependency; evolution; correctness; user satisfaction; uncertainty; discipline; control of assumptions; software engineering, methods, tools

mml453[papers]
17 November 1989

1 Fundamental Concepts of Software Engineering

The need for a software engineering discipline was first discussed at the international Garmisch Conference in 1968 [NAU69]. Since then many opportunities for progress have been conceived, investigated and, where appropriate, developed and introduced into practice. In general, such work addressed specific problems. The first major advance in program development was the introduction of, so called, high level **programming languages** (as distinct from machine languages) for program creation. This had, in fact, preceded the concept of a software engineering discipline by more than a decade. Increasingly such languages reflect concepts in which an application developer thinks and expresses himself. But industry has been slow in abandoning the older machine oriented languages whose direct use for program creation makes reliable and responsive program development and adaptation so much more difficult. With the increasing complexity of applications, their criticality in societal terms and, therefore, the ever growing need for dependability and adaptability transition to the use of languages appropriate to each application must be speeded up.

The move to high level, application oriented, languages was followed by the study of programming **methodology** [GRI78], recognition of the importance of **structure** and the development of a variety of structured and other programming **methods**. Application of these concepts led, in turn, to recognition of the need for **specification** of a program prior to its implementation. This eventually gave rise to pressures for wider use in program development of **formal** (mathematically defined), rather than natural, languages [JON80; TUR87]. Use of such languages had, heretofore, been largely restricted to the coding activity that had long been seen by many as the essence of *programming*. It now became apparent that major benefit was to be obtained from their application in program development activities that precede *coding*; problem **definition**, **requirements** analysis and **system specification** for example. It is these so called *up front* activities that ultimately determine operational characteristics. Yet they are, in general, cursorily treated in most software development projects. And even if undertaken but in a natural language, ambiguity is difficult to avoid, consistency cannot be demonstrated nor can completeness be systematically examined.

The concept of specifications and the development of formal languages for program development activities other than coding had originally emerged in academic circles from the realisation that the **correctness**, in some sense, of a program could and should be demonstrated by a process of **proof** based on rigorous mathematical argument [HOA69] or, even better, should be a **consequence** of a rigorous (mathematical) creation **process** [DIJ69]. **Testing**, the generally accepted means of demonstrating program acceptability can never demonstrate correctness of a program. It can only show that an error is present [DIJ72]. It was pursuit of this goal of constructive correctness that underlay the concepts of programming methodology, a developing discipline for ***programming in the small***.

Unfortunately, advances in programming methodology produced only limited benefit in the industrial development and evolution of large programming systems and of the larger systems in which embedded computers and their software play a controlling role. Development of such systems has become known as ***programming-in-the-large***. Slow progress in improving the technology employed in these areas was, in part, due to hesitation to impose the discipline inherent in adopting the rigorous approach to software development [JON80b] and to both real and imagined difficulties in so doing. More fundamentally, however, programming-in-the-large raised issues such as **variety** [BEL76], **uncertainty**, **complexity** and continuing **evolution** [BEL76; LEH80] that did not, in general, arise in the development of smaller programs and had, therefore, not been so extensively considered. The concepts and methods arising from studies of programming-in-the-small, while having an important contribution to make, did not address the major problems from which large system development has so long suffered.

The concepts of ***large programs***, of program dynamics and of their evolution had, however, been around since the late 1960s [LEH69]. Studies of these phenomena [BEL71, 72; LEH85] led inevitably to the realisation that the various stages of development over the lifetime of a software system interacted and influenced one another significantly. Local optimisation, for example, often leads to penalties at later stages of development and during subsequent usage. Conversely, a little extra directed investment and effort during one activity can subsequently yield significant benefit. Hence there has emerged the **process based** approach [SPW84, 86, 87] to software development already mentioned. This recognises that that process must be disciplined and addressed in its entirety, even if only to achieve some appropriate balance between global and local optimisation.

Recognition of the need to define and follow a disciplined **process** is perhaps the most important advance in system development of recent years. To achieve it one first needs a **process model** [LEH80, 85; SPW84, 86, 87] that defines a systematic and coherent path from formulation of an application concept *via* realisation of a usable system to its subsequent evolution. Process models may be generic or specific. Whether they can reasonably be considered algorithmic is a matter of some controversy [OST87; LEH87c]. Models are developed by first identifying technical and management activities required, the extent of information capture and storage, and the interfaces, relationships and dependencies between all these. Together they provide the structure and composition of the basic process. Given this, one then selects or develops **methods** to execute technical development activities. The introduction of defined and disciplined methods permits the application of computer based development **tools**. These provide mechanised support for individual activities and their systematic control. If appropriately conceived, the totality of methods and tools provides support for all aspects and stages of system evolution. True overall effectiveness will, however, be achieved only if data representation, methods and tools can and are **integrated** to provide full and **coherent** lifetime support. And even then, a process is only as good as is the rigour of its application. This is why techniques and tools to facilitate and control **planning** and **management** of a group and its activities, the **project**, must be included when planning and implementing integrated lifetime development support. Equally one requires both to support management of the emerging software and system **product** during development, its subsequent release to users and its evolution. Such product related function is exemplified by the need for management of component **variants** and **versions**, system **configuration** and **fault fixing**, and the control of system **evolution**.

For this great variety of tools to be able to interact and to support one another, and to achieve adequate support for application and product evolution, the collection must, as indicated above, be associated with an **information repository**. This retains all information relating to the development process, to the product produced and to the project that produces it for as long as it may be needed, some over the lifetime of the application. Facilities to support **communication** between machines and between people, **office** and **document preparation** facilities and so on are also required. If created as a coherent and integrated set, the resultant family of tools and services is termed an **Integrated Project Support Environment (IPSE)** [LEH87a, b].

The concepts and approaches outlined above reflect enabling technologies of an emerging software engineering discipline. Much of the associated technology is available for transfer to industry and the commercial world. But the rate of its penetration is too slow in relation to that at which computer systems are being introduced. To achieve an adequate rate of transfer and application in relation to the spread of computerisation represents a major challenge [LEH86]. An essential ingredient of any response is the increasing use by industry of software engineering expertise, in house or from appropriate external services. In particular industry must learn to understand the differing roles of programmers and software engineers [LEH86]. The former have responsibility for development of specific products; the latter for defining, developing and, perhaps, managing the process and its support. The difference between these roles is fundamental. Both must be supported if software that is to remain satisfactory over its entire lifetime is to be achieved. The titles given to those who fulfill these tasks may not be universally agreed [BLU89]. The fact is they are different and each is important in its own right. One must accept and willingly pay for both.

Following on the preceding discussion attention may be drawn to another fundamental property of software. Engineering technologies that have evolved in the past have provided development and management disciplines for artifacts embodied in physical form. The inventors, architects, designers and implementors of such artifacts have been controlled and constrained by **laws of nature**, by the **properties of the materials** processed and by the **visible** æsthetics of their production. Moreover, product evolution has occurred over hundreds if not thousands of years, has encouraged and been supported by the parallel development of mathematics and natural sciences, and has created the market forces that have led to product application. Science, technology and application have evolved in step.

Software technology is fundamentally different in each of these characteristics. It has evolved from primitive beginnings in a matter of decades. Mathematics provides a theoretical basis and a framework, contributing notation, techniques and methodology. A more general theory for software development has, however, been slow in developing relative to the growth in computer power and the spread of computers. ***The demand for software has outpaced the availability of an adequate technology to produce it***. Industrial development has been driven by market demand that is often uninformed and not discriminatory. The most distinctive feature of software development, however, is that it consists entirely of textual manipulation [LEH84b]. From first verbalisation of an application concept, whenever a system or a part of it is developed, fixed, enhanced, adapted or extended, text is added, changed and/or eliminated to achieve the desired result. In this process no natural laws, physical constraints or material properties operate. The manipulator is free, in a sense, to do what and as he pleases. The only constraining influences are the syntactic and semantic rules applying to the languages in which text is expressed at each step, pragmatic procedures and constraints that are introduced, and the enforcement of all these by adequate management and comprehensive mechanisation. The end result is visible only as a static textual model. That that model is satisfactory can be determined only by semantic interpretation or by examination of the results of execution. In addition, abstract artistic or mathematical and æsthetic judgements may be applied when text is perused. The more stringent the rules, procedures and judgements, the more do they provide an analogue of the constraints existing in the physical world and thereby the discipline demanded by the concurrent activity of many people over an extended time. Therein lies the real significance of formality, the introduction of method and the provision of supporting tools.

2 Uncertainty

2.1 Introduction to the Main Theme

The preceding overview of software engineering, an excerpt from a recently published paper [LEH89], provides a background for the main theme of this paper, an examination of the concept of *Program Correctness*. This is clearly an important issue in an age when society, individually and collectively, is becoming ever more dependent on computers and, therefore, on software. Issues such as the reliability of development and maintenance technologies, and the relative reliability of decisions by man and by machine cannot be addressed here. In passing it should, however, be noted that increasing use of method-based, often formal, program specification and development [JON80, TUR87] and of computer-based tools [STE85, LEH87a] represent major progress in both these areas. The question to be addressed here is more basic. Is there a limit to the confidence one may have in the correctness of the results of execution of programs that solve problems in the real world? In response to this question, three categories of *uncertainty* about the behaviour of such programs relative to the properties of their environment are outlined. In combination they lead to an Uncertainty Principle of Computer Application [LEH89]. A brief discussion of the role of software engineering in minimising uncertainty and its significance to society at large concludes the paper.

2.2 Program Classification

It has been proposed that programs may usefully be classified into three types [LEH80, 85a].

An *S-type* program is one for which the only criterion of acceptability is that *it satisfies some pre-stated and* (to be considered) ***absolute specification***. The specification is the sole, complete and definitive determinant of program properties. It is the only arbiter of the program being correct and satisfactory. The validity, relevance or æsthetics of the specification are extraneous issues, as are all program properties that are neither explicitly included in the specification or (formally) inferable from it.

A ***P-type*** program is one that has been created *to solve some stated problem*. The criterion for success is, here, that the solution obtained on execution is correct in a sense stated in or implied by the problem statement. If properties (side effects) not addressed in the problem statement are observed during program execution or otherwise, their implication on the correctness or satisfactory nature of the solution may be examined. If considered necessary, the statement must be modified and the program adapted and re-run to obtain an acceptable solution.

An ***E-type*** program is one developed *to solve a problem or implement an application in some real world domain*. The consequences of execution, the information conveyed to human observers, the behaviour it induces in attached or controlled artifacts, together determine its **acceptability**, **value** and the level of **satisfaction** it yields. Note that ***correctness*** has not been included amongst the criteria. That term should only be used to express a precise relationship based on calculable equivalence between a program or other formal representation and some higher level representation (specification) [TUR87]. The notions that replace the boolean concept of correctness are essentially fuzzy but may include quantitative as well as qualitative measures. Program acceptability depends on a **subjective** process of human assessment. ***It is the detailed behaviour under operational conditions that is of concern.***

The P-type program is intermediate between S and E-types and need not be separately considered in the present discussion. One may, indeed, usefully define an ***A-type*** program that is the union of P and E types. The modified classification schema bisects the universe of programs into those for which *correctness*, in the sense suggested above, is meaningful and those for which (at the whole system level) it is not.

2.3 The Process of Development

The parenthetical observation in the previous paragraph is of fundamental significance as will become clear from a closer examination of the software development process. Note that wherever the terms *process* or *development process* are used in this paper the reference includes both initial development and system or program evolution (colloquially termed *maintenance*)

The simple sequential process model [LEH84b] on which the IST ISTAR environment was based [LEH87a], its multi dimensional realisation and their predecessors [ZUR67; LEH85], all represent the process as a multi-step sequence. Each step involves *base* and *target* representations that may equally be referred to as *specification* and *implementation*. Only the first step, the first recorded verbalisation of the application, has no predecessor representation or specification. That role is played by the application in its domain, its objects, attributes, relations, events, activities. As mentioned above these exist, in general, in a continuous and unbounded domain and cannot be completely or precisely represented or even known.

The model or representation produced in the first step involves, therefore, either use, in part at least, of a non-formal representation or a major act of mental abstraction. The latter cannot be permitted since it is a transient and unobservable act that cannot be recorded, controlled or, in general, revisited. In the case of the former, a representation that is in part non-formal must be treated as if it were in its entirety non-formal since the consequences of ambiguities, incompatibilities and omissions that can be read into its non-formalised parts reflect into any formal elements of the complete representation. Hence the development process for E-type systems **is** and **must** be rooted in a non-formal representation and will display, at least some, of the characteristics of such systems.

Towards the end of the development process one obtains the solution system. Its most critical element, the executable code, is totally formal otherwise it could not serve as a computer *program*. As such it is essentially unambiguous and complete in relation to a given execution system (machine, peripherals and support software) though different machines might display different behaviour in execution. The final, operational, system is not, however and in general, entirely formal. It will, for example, include development and user documentation much of which will have to be in natural language. Such documentation is, in general, essential for successful usage. Its ambiguities or omissions can be a major source of misusage, unsatisfactory execution or results and so on. That is, the target system too, in its own right, must display some characteristics of non-formal systems. Note also that unlike the initial representation, this *final* representation is no exception to the rule stated above. In association with other material it becomes the specification of a further step, the next step of system evolution.

The process of E-type program development may, therefore, be described as *transformation of a non-formal representation of a real world application in its application domain to a formal representation of the programmatic part of a solution system in association with its non-formal support system operative in the real world solution domain.* The representation evolving during the transformation process will be a structure comprising many elements, sub-elements and so on. Some of these elemental representations will make the transition from non-formal to formal representation, each at an appropriate stage of the process. From the step that any element, when viewed as a specification, is formalised, *correctness* is applicable and must become the initial criterion of derived element acceptability, a judgement based on calculable criteria. Its semantics in terms of the behaviour of the element (in isolation) in execution can be known and is, in no way, uncertain. Each further step can produce an S-type element.

That is, all systems relating to the real world are of type E but the S-type program plays a major role in the development process. From the step in the process where a representation can be expressed in formal terms, that must be done [JON80b]. From then onwards the developer can concentrate on S-type programs. *Programmers* (in the conventional sense) *should never be given any program object other than an S-type to create.*

The objective of the development process, however, is to produce an artifact that serves some purpose in the real world. Therefore, after fulfilling all relevant verification obligations to demonstrate that a step has produced a *correct* target element, validation is also required [LEH84b]. Validation examines the real world implications of the formal semantics, determining the properties of each model in terms of properties not contained in or deducible from its specification. It should assess the element from three points of view. At the level of detail reached, the semantics of the model must satisfy the needs of the intended *purpose*. The model must also appear to define a satisfactory *solution system*. Finally it must be determined that, as the *specification* of the next step, the representation can be expected to provide a base for a viable continuation *process*. If, from one of these points of view, the validation process indicates an inadequacy of the current representation, it generates pressure for change in that representation, in its specification and, at least in principal, also in those of earlier steps. Where that representation is formal, processing of such a change must be itself also be fully formalised if S-type development is to be preserved.

2.4 Gödel Type Uncertainty

From the above it follows that an *E* type program may be described as a **model of a model · · · of a model of a computer application in the real world** [LEH80]. Turski regards each pair of neighbours in this chain of models as a *theory* and as a *model* of that theory respectively or, equally, as a *specification* and an *implementation* of that specification [TUR81]. At one extreme this view is reflected in his interpretation of the "two-legged" software-development process model [LEH84a]. Turski regards this as one in which a description of the real world application and the final implementation are both models of a *specification* that forms the bridge between concept and realisation. At the other extreme, the base and target representations at the core of each step of the canonical LST software (and systems) development process paradigm [LEH84b], for example, also form a theory and model pair.

It follows that every *E*-type program is Gödel incomplete [GÖD31], an instance of Bondi's more general observation [BON77]. The properties of such programs cannot be completely known from within the system. Now those involved in system development and usage become an integral part of the system. Their mental activities direct and drive the *process* of system development and evolution. The degree of satisfaction that the operational system yields is ultimately determined by the action and inaction of designers and users. Thus neither developers nor user can fully know system properties. Gödel incompleteness is transformed into *Gödel*-type uncertainty [LEH89]. The process must seek to limit this uncertainty to representational incompleteness.

2.5 Heisenberg Type Uncertainty

A second form of uncertainty arises from system development and operation. As a consequence of the execution of the development process, understanding of the application changes. User ambitions are stimulated. Alternative methods of solution are recognised. Apparent opportunities for improvement abound. Moreover, the system must evolve. Anyone using computers seriously will have experienced the continuing *maintenance* that the associated software appears to require. This is not entirely due to shortsightedness on the part of either users or developers. All artificial systems must evolve [SIM69] if they are to remain satisfactory; but the rate of computer system evolution needs to be substantially greater [LEH85]. One source of the continuing pressure for change in *E*-type systems is that each system includes an implicit model of itself. Moreover, the application, its domain and human perception of both change. It is the associated feedback that drives system evolution. Software adaptation is the primary means whereby such evolution is achieved.

Intrinsic delay between recognition and implementation of a need or opportunity for change means that mismatch between human desires and system properties cannot be permanently eradicated. Satisfaction with the system declines unless the software (which largely determines system properties) is repeatedly updated (first law of program evolution [LEH74, 85]). Moreover, as the system evolves, change implementation requires combinatorially increasing relative effort (second law of program evolution [LEH74, 85]). Thus the more precise knowledge is of the application, its solution and their respective domains, the less is it possible to maintain satisfactory system behaviour and satisfactory delivered results. The source of dissatisfaction is a function of perception and understanding, and cannot be completely predicted. Hence the system displays *Heisenberg*-type uncertainty [LEH77]. Here too, the process and its management is key to minimisation of the consequences of this inherent uncertainty.

2.6 Pragmatic Uncertainty

There is also a third type of uncertainty [LEH89]. The domain of an *E*-type application is, in general, unbounded, effectively continuous and dynamic, always changing. The solution system is finite, discrete and, in the absence of human intervention, static. The process of deriving one from the other involves a variety of *assumptions* about the application, its domain, perceived needs and opportunities, human responses to real world events, computational algorithms, theories about all these and so on. Some assumptions will be explicitly stated, others will be implicit in the design and implementation detail. All will be built into the final system.

In a dynamic world the facts on which any assumption set is based will be modified by system-exogenous events. However carefully the validity of assumptions is controlled when adopted, some will be less than fully valid when the results of execution are used. But this is when, to be fully satisfactory, a program needs to be correct. Correctness of a program specification and its derivation is a means to that end, necessary but not sufficient. The assumption set must be maintained correct by appropriate changes to program or documentation texts in a time frame determined by the application and the nature of the required change. This is impossible even if all assumptions were explicit and their location precisely known. *Pragmatic* uncertainty in computer system behaviour is inevitable. It is intrinsic to mechanised computation and is intimately linked to Heisenberg-type uncertainty.

2.7 An Uncertainty Principle

This analysis leads to an Uncertainty Principle for Computer Application:

The outcome, in the real world, of software system operation is inherently uncertain with the precise area of uncertainty also not knowable [LEH89].

Gödel-type uncertainty is, primarily, a matter of formalism, of theoretical interest but unlikely to have significant practical implications. The consequences of the Heisenberg-type is the never ending maintenance burden that accompanies all serious computer usage. Pragmatic uncertainty is the most challenging. It leads to concerns that must increase as computer based systems become larger, more complex and more intimately interwoven with the life and activity of individuals and of society at large. It is this which is of most concern to the present workshop. Research and development in programming methodology and software technology is providing methods and tools to ensure that programs can be satisfactorily developed and maintained. This work must continue and processes that exploit the results introduced into general industrial practice.

3 Control of Uncertainty Through the Engineering of Software

The principle just stated leads to immediate, practical conclusions. The joint responsibility of user, software engineer and programmer (in the very wide role as outlined at the end of section 1 to include those who, for example, undertake activities often described as systems analysis, design, coding integration and so on) is to reduce, ultimately to minimise, the encounter with uncertainty or the consequences of uncertainty.

The user is primarily responsible for defining the application and the application domain. In doing this he must seek to ensure completeness of his discussion and analysis including consideration of the likely impact of system installation and operation. Such prediction is no mean task especially in view of the close and intimate coupling between the system and its users, individually and collectively. The likely average reaction of those interacting with the system in usage is perhaps predictable. The specific reactions of individuals to particular situations is not; yet it may have dramatic consequences. Equally the user must consider the nature and likelihood of change in the the operational environment or in the goals of the application, the possible consequences if incompatibilities arise between the system and the application domain as it is at the time of execution. The user is the ultimate arbiter for the validity of assumptions embedded in the system and must accept responsibility for the consequence of decisions which ultimately become no longer valid.

Programmers have involvement in and responsibility for all stages of definition, design and implementation of software systems as per the wider definition summarised above and discussed previously [LEH86]. In this capacity uncertainty has direct implications. In the first instance, and throughout the development and lifetime of the system, they must consider, record and take account of the user's observations, definitions and forward looking perspectives. As decisions are taken and assumptions consciously made or implied by analysis, design and implementation activity, they must be captured and faithfully recorded. Above all they must accept the discipline and constraints imposed to assure not only initial correctness but to make possible the subsequent evolutionary adaptation of the system to changing circumstances.

One facet of the role of the software engineer is as a *process engineer* developing, evaluating or introducing processes, methods and tools into practice. Another calls for involvement with development or maintenance of a specific product as process manager or support engineer. Each role is influenced by the need to take account of the inherent uncertainty associated with software development and the maintenance of user satisfaction. In specifying, evaluating and acquiring or developing methods and tools, significant emphasis must be placed on the need to highlight, capture and record assumptions in retrievable fashion, whatever their basis or nature. Linkages must therefore be provided in all relevant tools that alert the participants in the programming process whenever certain actions are taken to provide an appropriate *pointer* to any such assumption. It is not possible to present here an exhaustive analysis of the circumstances when such action is desirable or necessary. An indication of the breadth of concern is provided by a list that includes such varied activities as the choice of factors to be considered and to be excluded from consideration in the implementation, the adoption of specific theories covering some aspect of the application, the selection of algorithms, design decisions of any sort, the fixing of branch conditions, the adoption of computational procedures, assignment of values to data and constants and so on.

The discussion of the previous paragraph primarily addressed the issues raised by *pragmatic uncertainty* that causes *software pollution* through the gradual erosion of assumption validity. Consideration of Heisenberg type uncertainty, while interacting strongly with the former, focuses attention more specifically on the development process; its structure, activities rigour, degree of mechanisation and support. The present paper can only provide an introductory overview, indicating that control of evolution driven by this class of uncertainty is a major element of the software manager's task. Not for nothing has software management been long described as *the management of change*. And in many ways this might be seen as the prime responsibility and opportunity for software engineering and the software engineer.

Simply summarised, the process of development and evolution must be designed, not only to produce a product that has the desired, even optimum, attributes at the time of delivery. It must ensure that the initial satisfaction is maintained and even enhanced throughout the lifetime of the system. All stages of the process must be designed (methods and tools) to identify and record areas in which changes may occur and those in which, if changes occur, the consequences to the user or to adaptive action are non-trivial. It must also provide and integrate management decision and control procedures to ensure that appropriate procedures are adopted and practised with regard, for example, to change authorisation, planning, execution and installation. This, in turn, implies that the necessary information is available or drawn to the attention of both the manager and those involved in technical implementation. The problems of information capture, retention and retrieval become central and one in which rule and knowledge based systems find important application. Recognition of the above process needs is not new. Their unification in the context of the uncertainty theory merely reinforces what has previously been recognised and adopted. In association with other necessary steps not discussed here, they present a major challenge for the continued refinement and extension of the software development and evolution process, and the methods and tools that are its building blocks. Software engineering faces a major opportunity and challenge.

Gödel type uncertainty poses somewhat different, perhaps less immediate, challenges. It must not be ignored since it has an impact on modelling and achieving complete understanding of the process. In particular, its implications may arise in the the design and support of formal aspects of the process. The software engineer must, however, have at least an understanding of the problem and when, if at all, it must be taken into consideration

4 Conclusions

In its main section this paper has presented what may at first encounter appear as a largely philosophical and theoretical result of little consequence. It is to be hoped that the brief introduction to the challenges this result poses to software engineering and to ways and means whereby this challenge may, indeed must, be met, has convinced the reader otherwise.

An alternative reaction may be to say that most, if not all, of what has been said is widely known. It has, undoubtedly, long been recognised that, in general, the quality of industrial software development and of its product falls far short of the levels of reliability demanded by the expanding range of computer application. Concern was first publicly expressed at the Garmisch International Workshop on Software Engineering [NAU69]. Dijkstra [DIJ69, 72], Hoare [69, 71], IFIP programming methodology working group WG 2.3 [GRI78] and many others have been stressing the need for program correctness for over two decades. Debate on SDI led by Parnas [PAR85] and others, and on the uses and limitations of proof techniques [DEM79, VAR79] and program verification [FET88, VAR89], reflect strong concern within the wider computing community. Neumann [NEUM], Thomas [THO89] and others are exposing the limitations of computer control in life-critical applications. Each of these authors discusses specific aspects of the problem of reliable software development. Reliability is here used in the sense that the product will produce satisfactory results whenever executed in its subsequent lifetime. Their concern is not confined to academic circles. Awareness of the problems experienced in software development as reflected in the management of the development process is widespread in industry. In most instances these have been attributed to human failure. "Why can't programmers be like other Engineers?" is a frequently heard plaint in industry.

It must be recognised that the introduction and continued application of software dependent computer systems faces problems not relevant in other disciplines [LEH85, 89]. Real and fundamental differences exist between the development and adaptation (evolution) of software and that of physical artifacts. Yet, as computers are applied ever more widely, the implications of these differences have not been sufficiently considered in seeking to improve the industrial process whereby applications are implemented.

Further problems arise from the need to embed assumptions about an ever changing application environment in the software, change that is accelerated through creation of the software and use of the computer. It is the thesis of this paper that uncertainty in the detailed properties of software and its behaviour when executing and, therefore, in computer application is inescapable. This fact is a challenge to society in general, to prime movers in computer application, to implementors and to supporting software engineers in particular. The first become involved when they select and authorise the applications and determine the properties of the end system; the latter two classes because they conceive, control and execute the process of implementation. Uncertainty will always be there. It is the responsibility of prime movers that society is not unnecessarily exposed through thoughtless application. To avoid this, they and society at large must be informed of the threat. It is the responsibility of software engineers and implementors that the public are informed, and to ensure that risks associated with uncertain behaviour and the consequences, should the unanticipated be encountered, are minimised. Rigorous enforcement of advanced software technology, systematic application of disciplined methods and mechanisation can make a fundamental contribution to this end. Their widespread, if not universal, adoption must be accepted as an urgent societal priority.

Relative to the concerns expressed, some aspects of the phenomena discussed in the present paper may appear to have little practical significance. This is not so. These phenomena and issues related to them reflect basic and intrinsic properties of the process of computer program development, usage and evolution, an activity that intimately effects the whole of mankind. They provide a unifying concept to manifold observations with concepts and components of a theory of software development. The latter is essential for further steady progress towards a reliable, responsive and cost effective technology for the development of computer based systems that can be commissioned with every confidence that they will provide continuing satisfactory service.

Computer usage is penetrating ever deeper into the very fabric of society. The dependence of mankind on the correctness of computer systems in general, and on software in particular, becomes ever greater. Correctness is, primarily, a relationship between the state of the application domain at the time when the results of a computation are applied and the software which prescribes the computation. The technology, extension and wide application of formal methods to maximise the opportunity for initial correctness over the process stages where they can be applied is a most significant first step. To ensure continuing safety in system usage it must be accompanied by the wider use of advanced software engineering technology. That discipline, as reflected in the software development and evolution process, in the methods used and in process mechanisation, is the means whereby uncertainty and the consequences of uncertainty can be minimised and user satisfaction maintained. Above all, however, emphasis must be placed on *responsibility*, *conscientiousness* and *care* in the selection, definition, development and control of computer applications. This is a matter that concerns not only computer scientists and software engineers. It is a matter for industry, for government, for all levels of the educational system, indeed for all of mankind.

Given the increasing dependence of mankind on software based systems it is a matter of urgent priority to cope with these issues. Strict discipline and mechanisation through the application of advanced software engineering in all its aspects offers a practical solution, reducing uncertainty to the level at which it is present in all human activity. The time for such introduction is ripe and the paper has pointed to ways in which it may be achieved.

5 References

[BEL71] Belady L A and Lehman M M, *Programming System Dynamics or the Metadynamics of Systems in Maintenance and Growth*, IBM Res. Rep. RC 3546, Sept. 1971, T J Watson Res. Ctr., Yorktown Hts, NY, 10598

[BEL72] id., *An Introduction to Program Growth Dynamics*, in *Statistical Computer Performance Evaluation*, W Freiburger (ed), Academic Press, New York, 1972, pp. 503 - 511

[BEL76] id., *A Model of Large Program Development, IBM Sys. J. vol. 15, no. 3, pp. 225 - 252*

[BON77] Bondi H, *The Lure of Completeness, The Encyclopedia of Ignorance*, R Duncan and M Weston-Smith (eds.), Pergamon Press, London, 1977, pp. 5 - 8

[BLU89] Blum B I, *Volume, Distance, and Productivity*, J.of Sys. & Softw., vol. 10, no. 3, Oct. 1989, pp. 217 - 226

[DEM79] DeMillo R A, Lipton R J and Perlis A J, *Social Processes and Proofs of Theorems and Programs*, CACM, vol. 22, no. 5, May 1979, pp. 271 - 280

[DIJ69] Dijkstra E W, *A Constructive Approach to the Problem of Program Correctness*, BIT, vol. 8, no. 3, 1969, pp. 174 - 186

[DIJ72] id., *The Humble Programmer*, ACM Turing Award Lect., CACM, vol. 15, no. 10, Oct. 1972 pp. 859 - 866

[FET88] Fetzer, J.H, *Program Verification: The Very Idea*, CACM, vol. 32, no. 8, Sept. 1988, pp. 1048 - 1063

[GÖD31] Gödel K, *Über Formal Unentscheidbare Sätze der Principia Mathematica und Verwandter Systeme*, Monatsheftt für Mathematik und Physik, vol. 38, pp. 173 - 198

[GRI78] Gries D, *Programming Methodology - A Collection of Articles by Members of IFIP WG2.3*, Springer Verlag, New York, 1978

[HOA69] Hoare C A R, *An Axiomatic Basis for Computer Programming*, CACM, vol. 12, no. 10, Oct, 1969, pp. 576 - 583

[HOA71] id, 'Proof of a Program FIND', CACM, vol. 14, no. 1, Jan. 1971

[JON80a] Jones C B, *The Role of Formal Specifications in Software Development*, InfoTech State of the Art Conf. on Life Cycle Management, Report se. 8, no. 7, 1980, inv. papers, pp. 117 - 133

[JON80b] id., *Software Development - A Rigorous Approach*, Prentice - Hall Inc., New York,1980

[LEH69] Lehman M M, *The Programming Process*, IBM Res. Rep. RC 2722, IBM Res. Centre, Yorktown Heights, NY 10594, Sept. 1969 and in [LEH85], pp. 39 - 84

[LEH74] id., *Programs, Cities, Students - Limits to Growth*, Imperial College. Inaugural Lecture Series, vol. 9, 1970 - 1974. Also [GRI78], pp. 42-69 and [LEH85], pp. 133 - 163

[LEH77] id., *Human Thought and Action as an Ingredient of System Behaviour*, *The Encyclopedia of Ignorance*, R Duncan and M Weston-Smith (eds.), Pergamon Press, London, 1977, pp. 347 - 354

[LEH80] id., *Programs, Life Cycles and Laws of Software Evolution*, Proc. IEEE Special Issue on Software Engineering, Sept. 1980, pp. 1060 - 1076

[LEH84a] id., *A Further Model of Coherent Programming Models*, in [SPW84], Feb. 1984, pp. 27 -35

[LEH84b] Lehman M M, Stenning N V and Turski W M, (1984). *Another Look at Software Design Methodology*, ICST DoC Res. Rep. 83/13, June 1983 and Software Engineering Notes, vol. 9, no 2, April 1984, pp. 38 - 53

[LEH85] Lehman M M and Belady L A, *Program Evolution - Processes of Software Change*, Academic Press, London, 1985

[LEH86] Lehman M M, *Advanced Software Technology - Development and Introduction to Practice*, Invited Paper, Information Processing '86, Proc. IFIP Congress 1986, Dublin, September 1-5, Elsevier Science Publishers (BV), (North Holland), pp. 605 - 661

[LEH87a] Lehman M M, *Model Based Approach to IPSE Architecture and Design - The IST ISTAR Project as an Instantiation*, Inv. Contr., Quart. Bul., IEEE Comp. Soc. Tech. Comm. on Database Eng., Sp. Iss. on Softw. Eng. System and Database Reqs., vol. 10, no. 1, 1987, pp. 2 - 13

[LEH87b] Lehman M M and Turski W M, *Essential Properties of IPSEs*, Software Engineering Notes, vol. 12, no. 1, pp. 52 - 55

[LEH87c] Lehman M M, Process *Models, Process Programs, Programming Support - Invited Response To A Keynote Address By Lee Osterweil*, Proc. 9th Int. Conf. on Softw. Eng., Monterey, CA, 30 March - 2 Apr. 1987, IEEE Comp. Soc. pub. no. 767, IEEE Cat. no. 87CH2432-3, pp. 14 - 16

[LEH89] id., *Uncertainty in Computer Application and its Control Through the Engineering of Software*, in Software Maintenance: Research and Practice vol. 1, no. 1, Sept. 1989, John Wiley & Sons Ltd, London and New York, pp. 3 - 27

[NAU69] Nauer P and Randell B, *Software Engineering - Report on a Conference Sponsored by the NATO Science Committee*, Garmisch, 1968, Scientific Affairs Division, NATO, Brussels 39, 1969

[NEUM] Neumann P G (ed), *Risks to the Public in Computers and Related Systems* regular feature in every issue of Software Engineering Notes, Special Interest Group on Software Engineering, ACM Press, ACM, NY, NY 10036

[OST87] Osterweil L, *Software Processes are Software Too*, Proc. 9th Int. Conf. on Softw. Eng., Monterey, CA, 30 March - 2 Apr. 1987, IEEE Comp. Soc. Pub. no. 767, IEEE Cat. no. 87CH2432-3, pp. 2 - 13

[PAR85] Parnas D, *Software Aspects of Strategic Defense Systems*, American Scientist, vol. 75, no. 3, Sept. - Oct. 1985, pp. 432 - 440. Revised version in Comm. ACM. vol. 28, no. 12, Dec. 1985, pp. 1326 - 1335

[SIM69] Simon HA, *The Sciences of the Artificial*, M.I.T. Press, Cambridge, MA. 1969, 2nd ed. 1981

[SPW84] Potts C (ed), *Proceeding of the Software. Process. Workshop*, Egham, Surrey, UK., Feb. 1984. IEEE, cat. no. 84CH2044-6 Comp. Soc., Washington D.C., order no. 587

[SPW86] Wileden J C and Dowson M (eds), *SE Notes Special Issue on the 2nd International Workshop on the Software Process and Software Environments*, Coto de Caza, Cal., 27-29 March 1985, vol. 11, no. 4, Aug. 1986

[SPW87] Dowson M (ed), *Iteration in the Software Process*, Proceedings of the 3rd International Process Workshop, IEEE Comp. Soc. Press, March 1987

[STE85] Stenning N V, *Software Engineering: Present* and Future in *The Corporate Database, State of the Art Reports*, D Iggulden (ed), se. 13, no. 3, Pergamon Infotech Ltd, Maidenhead, England, 1985, pp. 83 - 93

[THO89] Thomas, M, *Development Methods for Trusted Computer Systems*, BCS Annual Lecture, in Formal Aspects of Computing, vol. 1, no. 1, 1989, pp. 5 - 18

[TUR81] Turski W M, *Specification as a Theory with Models in the Computer World and in the Real World*, Infotech State of the Art Report, se. 9, no. 6, 1981, pp 363 - 377

[TUR87] Turski and Maibaum T, *The Specification of Computer Programs*, Addison Wesley, London, 1987

[VAR79] Various correspondents, *Comments on a Paper by De Millo, Lipton and Perlis*, [DEM79], Comm. ACM, vol. 22, no. 11, Nov. 1979, pp. 621 - 630

[VAR89] Various correspondents, *Comments on a Paper by Fetzer*, [FET88], Comm. ACM, vol 32, no. 3, March 1989, pp. 287 - 290 and pp. 374 - 381

[ZUR67] Zurcher F W and Randell B, *Iterative Multi-Level Modelling - A Methodology for Computer System Design*, IBM Res. Rep. RC 1938, Nov. 1967, IBM Res. Centre, Yorktown Heights, NY 10594 and *Information Processing '67*, Proc. IFIP Congr. 1968, Edinburgh, Aug. 1968, pp. D138 - 142

VIEWGRAPH MATERIALS

FOR THE

M. LEHMAN PRESENTATION

SOFTWARE, SYTEMS & APPLICATION UNCERTAINTY
&
ITS CONTROL THROUGH THE ENGINEERING OF SOFTWARE

SEL SOFTWARE ENGINEERING WORKSHOP

29 NOVEMBER1989

M M LEHMAN
LEHMAN SOFTWARE TECHNOLOGY ASSOCIATES LTD
60 ALBERT COURT
PRINCE CONSORT RD
LONDON SW7 2BH
&
DEPARTMENT OF COMPUTING
IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY& MEDICINE
180 QUEEN'S GATE
LONDON SW7 2BZ

PROGRAM CLASSIFICATION

- *S*-TYPE
  - COMPLETELY DEFINED BY ***SPECIFICATION***
  - CRITERION OF SUCCESS IN IMPLEMENTATION
    ⇒ ***CORRECTNESS*** RELATIVE TO SPECIFICATION
    ⇒ ***PROVABLY*** OR BY VIRTUE OF ***DERIVATION***

- *P*-TYPE
  - SOLVES SPECIFIC ***PROBLEM***
  - CRITERION OF SUCCESS IN IMPLEMENTATION
    ⇒ ***SATISFACTORY*** (CORRECT?) SOLUTION
    ⇒ 'SATISFACTORY' MUST BE ***DEFINED***

- *E*-TYPE
  - ITS USE REALISES APPLICATION IN REAL WORLD
  - CRITERION OF SUCCESS IN IMPLEMENTATION
    ⇒ USER ***SATISFACTION***
    ⇒ ON EACH ***APPLICATION***, THAT IS,
    WHENEVER RESULTS ARE ***USED***

NOTE - P & E TYPES MAY BE COMBINED INTO UNIFIED *A - TYPE* THIS BISECTS PROGRAM CLASSES INTO THOSE THAT MUST BE ***CORRECT*** & THOSE THAT MUST BE ***SATISFACTORY*** AT THE TIME WHEN APPLIED

THE NEED FOR CONTINUING CHANGE

- USER *SATISFACTION* CRITERION OF SUCCESS
- INITIAL CORRECTNESS ***NECESSARY*** TO ACHIEVE IT. PROGRAM **ELEMENTS** ***SHOULD***, **THEREFORE,** BE CONSTRUCTED FROM *S*-TYPE ELEMENTS
- CORRECTNESS ALONE ***NOT SUFFICIENT***
- RESULTS OF EXECUTION MUST BE ACCEPTABLE WHEN ***APPLIED***
- REAL WORLD DYNAMIC, UNDERGOING CONTINUING **CHANGE**
- SOFTWARE IS A MODEL OF THAT WORLD WITH BUILT IN, POSSIBLY IMPLICIT OR HIDDEN, ASSUMPTIONS
- RESULTS OF EXECUTION WILL REFLECT THESE
- ***VALIDITY*** OF SOME, EVEN IF INITIALLY JUSTIFIED, MUST HAVE CHANGED SINCE BEING EMBEDDED
- SYSTEM SHOULD UNDERGO CONTINUING *CHANGE*

CONTINUING EVOLUTION

- PROGRAM **DEVELOPMENT, INSTALLATION, USE & EXOGENOUS CHANGE,** ***MODIFY***:
  - ***APPLICATION***
  - APPLICATION ***DOMAIN***
  - ***PERCEPTION*** OF BOTH
  - ***UNDERSTANDING*** OF ***PROBLEM***
  - BASIS OF & JUSTIFICATION FOR ***ASSUMPTIONS***
  - AND OF POSSIBLE APPROACHES TO ***SOLUTION***
  - ***NEEDS, OPPORTUNITIES, AMBITIONS***
  - ***TECHNOLOGY***

- **CRITERIA** FOR PRODUCT ACCEPTABILITY ALSO CHANGE

- DRIVEN BY
  - ***FEEDBACK***
  - ***EXTERNAL*** CHANGE

- EVOLUTION **INTRINSIC** TO COMPUTER ***APPLICATIO***N,

- ***SYSTEMS*** MUST BE CONTINUOUSLY EVOLVED TO ADAPT THEM TO ***CHANGING*** APPLICATION DOMAIN & TO CHANGING ***VIEWS*** OF THAT DOMAIN

SOFTWARE THE MEANS

- **PROCESS** MUST **SUPPORT & CONTROL** EVOLUTION

FIRST LAW OF PROGRAM EVOLUTION:-

E-TYPE PROGRAMS MUST BE CONTINUALLY CHANGED ELSE THEY DECLINE IN USEFULNESS & IN THE SATISFACTION THEY DELIVER

- *UNIVERSAL* EXPERIENCE AS ILLUSTRATED BY LIFE CYCLE *COSTS*
  - INITIAL DEVELOPMENT < *30%*
  - EVOLUTION (MAINTENANCE) > *70%*

- **MAINTENANCE TO PRESERVE**
  - USER ***SATISFACTION***
  - VALIDITY OF ***ASSUMPTION SET***

- **MAINTENANCE AS CONTINUING ADAPTATION**

MAINTAINING MODEL RELATIONSHIP BETWEEN REAL WORLD & SOFTWARE

SOFTWARE AS MODEL OF REAL WORLD

REAL WORLD	SOFTWARE
• INDEPENDENT EXISTENCE	• MODEL
• NATURAL LAWS	• THEORIES MODELS
• EFFECTIVELY UNBOUNDED	• BOUNDED
• EFFECTIVELY CONTINUOUS	• DISCRETE
• DYNAMIC	• STATIC
• PHYSICAL	• TEXTUAL
• CONCRETE	• ABSTRACT
• AT MOST, INFLUENCED BY	• PART OF

SOLUTION SYSTEM

IN DEVELOPING SOFTWARE MODEL THAT LARGELY DETERMINE SYSTEM FUNCTIONAL CHARACTERISTICS *ASSUMPTIONS* PLAY KEY ROLE

NATURE OF REAL WORLD

- **INDEPENDENT** EXISTENCE THAT INCLUDES CHANGING, POSSIBLY **UNPREDICTABLE**, ELEMENTS

- EFFECTIVELY **UNBOUNDED** (INFINITE)
  - TYPES, NUMBERS OF
    - ⇒ **ATTRIBUTES**
    - ⇒ **STRUCTURES**
    - ⇒ **PROCESSES**
  - **METRIC** PROPERTIES

- EFFECTIVELY **CONTINUOUS**
  - **ENTITIES**
  - **STRUCTURES**
  - **PROCESSES**
  - **DATA**

- **DYNAMIC**
  - ALWAYS CHANGING

- **NATURAL** LAWS WHICH CAN BE APPROXIMATED BY *MODELS*, *PERHAPS CONTROLLED*,BUT *NOT CHANGED*

- **CONCRETE**
  - CAN BE OBSERVED
  - EXPERIENCED
  - MEASURED

- **PHYSICAL** PROPERTIES
  - CONSTRAINTS
  - LIMITS

NATURE OF SOFTWARE

- *MODEL* OF **APPLICATION** IN ITS ***DOMAIN***
- MODEL OF **SOLUTION** IN ITS DOMAIN
- PART OF **SOLUTION SYSTEM** IN ITS DOMAIN
- **STRICTLY BOUNDED** (FINITE)
  - **ATTRIBUTES**
  - **STRUCTURES**
  - **ALGORITHMS**
  - **DATA** REPRESENTATIONS
- **DISCRETE**
  - **ENTITIES**
  - **STRUCTURES**
  - **PROCESSES**
  - **DATA**
- **STATIC**
  - **HUMAN** INTERVENTION FOR CHANGE
- **BUILT IN THEORIES**
  - **REFLECT** (MODEL) LAWS
  - **MATHEMATICAL** PROPERTIES
- **ABSTRACT**
  - CAN ONLY BE ***UNDERSTOOD***
  - ***MATHEMATICAL*** MANIPULATION FOR FORMALISED REPRESENTATIONS
- **TEXTUAL REPRESENTATION**
  - **SEMANTICS**
  - **SYNTACTIC** CONSTRAINTS
  - **PRAGMATIC** LIMITS, IF ANY?

IN DEVELOPING SYSTEM, *ASSUMPTIONS* PLAY KEY ROLE

NEED FOR ASSUMPTIONS

- **FINITISATION**
  - PHENOMENOLOGICAL
  - COMPUTATIONAL
- **DISCRETISATION**
  - PHENOMENOLOGICAL
  - COMPUTATIONAL
- **ABSTRACTION**
  - SELECTION
  - DISCARDING DETAIL
- ADOPTION OF **THEORIES**
- DEVELOPMENT OF **PROCEDURES**
- **DATA**
  - VALUE
  - RATE OF CHANGE
  - DEPENDENCIES
- DEVELOPMENT OF **MODELS**
  - PHENOMENOLOGICAL
  - OPERATIONAL
  - MANAGERIAL
  - COMPUTATIONAL
  - PROCEDURAL
- **ALGORITHMS**
  - SELECTION
  - REALISATION
- **RELATIONSHIPS**
- **etc., etc.**

ASSUMPTIONS

RELATE TO
- *REAL WORLD*
- *APPLICATION*
- *USERS*
- *COMPUTATIONAL PROCEDURES*
- *SOFTWARE & SYSTEM*
- *EMBEDDED DATA*
- *ASSOCIATED DATA BASE*

- ESTIMATE *ONE* REAL-WORLD ASSUMPTION FOR EVERY *TEN* LINES OR SO OF PROGRAM CODE

- IN **LARGE** PROGRAM THERE **MUST** BE ASSUMPTIONS OF QUESTIONABLE VALIDITY

- MANY WILL BE ***IMPLICIT***

- *SOONER OR LATER* SOME, **THEN** INVALID OR MISSING, ASSUMPTION OR DATA WILL ***CAUSE PROBLEM***

  BECAUSE OF
  - FREQUENCY OF EXECUTION
  - SPEED OF EXECUTION
  - TIGHTNESS OF USER COUPLING

- RESULTS OF EXECUTION MUST BE CORRECT WHEN USED

- **CHANGES** CANNOT BE MADE **INSTANTANEOUSLY**

WHAT ARE IMPLICATIONS OF THESE FACTS OF LIFE?

CONCEPTUAL IMPLICATIONS

- *ALL* ASSUMPTIONS SHOULD BE MADE EXPLICIT
  CAPTURED, RECORDED & MAINTAINED VALID

- *DATA* (VALUE , RANGE, TYPES) MUST BE REVIEWED
  AS APPROPRIATE, BEING TIME & EVENT DEPENDENT

- FAILURE TO DO SO CREATES ***SOFTWARE POLLUTION***
  - ⇒ ***UNCERTAINTY*** IN APPLICATION
  - ⇒ ASSOCIATED ***RISK***

- **MAINTENANCE** ⇒ USER SATISFACTION
  - ⇒ ASSUMPTION SET

- NEED FOR ***ALERTNESS, REVIEW, UPDATING***
  - ⇒ ***CONTINUING*** EVOLUTION

- PROFESSIONAL ***RESPONSIBILITY*** TO SOCIETY
  - ⇒ ***CONTROL*** OF APPLICATIONS
  - ⇒ ***SPECIFICATION & DEVELOPMENT***
  - ⇒ ***EVOLUTION***
  - ⇒ MINIMISATION OF ***RISK OF***
    - ⇒ ENCOUNTER WITH INVALID OR INCOMPLETE ASSUMPTIONS
    - ⇒ CONSEQUENCES SHOULD IT OCCUR

- INTRINSIC ***UNCERTAINTY*** *IN* ***E-TYPE*** *SOFTWARE*

  & HENCE

- *IN* ***COMPUTER APPLICATION*** *IN* ***REAL WORLD***

AN UNCERTAINTY PRINCIPLE

THE OUTCOME, IN THE REAL WORLD, OF **E-TYPE** ***SOFTWARE SYSTEM OPERATION IS INHERENTLY UNCERTAIN WITH THE PRECISE AREA OF UNCERTAINTY ALSO NOT KNOWABLE***

TYPES OF UNCERTAINTY

GÖDEL

- *E*-TYPE PROGRAM IS A MODEL OF A MODEL ··· OF A MODEL OF AN APPLICATION IN REAL WORLD
- EACH MODEL PAIR CAN BE INTERPRETED AS A THEORY & A MODEL OF THAT THEORY OR AS A SPECIFICATION & ITS IMPLEMENTATION
- HENCE EVERY PROGRAM IS GÖDEL INCOMPLETE
- GÖDEL-TYPE APPLICATION UNCERTAINTY IS A REFLECTION OF GÖDEL INCOMPLETENESS

HEISENBERG

- SYSTEM DEVELOPMENT, INSTALLATION & USE CHANGES APPLICATION, SOLUTION, PERCEPTION & UNDERSTANDING OF THESE
- THE MORE PRECISE KNOWLEDGE IS OF THE APPLICATION & ITS SOLUTION THE LESS WILL THE RESULTS OF EXECUTION SATISFY USER
- THE SOURCE OF NON-SATISFACTION, BEING A FUNCTION OF PERCEPTION & UNDERSTANDING, CANNOT BE PREDICTED

PRAGMATIC

- VALIDITY OF THE TOTALITY OF EMBEDDED ASSUMPTIONS, EXPLICIT & IMPLICIT, CANNOT BE KNOWN OR MAINTAINED RESPONSIVELY,
- RESULTS OF EXECUTION ARE, THEREFORE NOT COMPLETELY PREDICTABLE

- THERE IS CLOSE RELATIONSHIP BETWEEN THE HEISENBERG & PRAGMATIC TYPES OF UNCERTAINTY

PRACTICAL IMPLICATIONS

- *AWARENESS* OF PROBLEM, AVOIDING IMPLICIT ASSUMPTIONS
- *ADOPTION & IMPLEMENTATION* OF ASSUMPTIONS MUST BE CONTROLLED, RECORDED & REVIEWED
- DITTO FOR ***DATA***
- ANTICIPATE & IDENTIFY ***POTENTIAL CHANGES & CHANGE SENSITIVE*** AREAS
- SYSTEMATIC, PERIODIC, DETAILED SYSTEM WIDE REVIEW BY JOINT USER/IMPLEMENTATION TEAMS
- DISCIPLINED, CONTROLLED, RECORDED **EVOLUTION** (MAINTENANCE)
- TOTAL PROCESS MECHANISATION WITH ***ACTIVE PROCESS SUPPORT & GUIDANCE***
- ***EDUCATION & FAMILIARISATION*** OF SOCIETY

VIEWPOINT

- SOFTWARE AN **ORGANISM** *NOT* AN ARTIFACT

- GROWS & EVOLVES THROUGH **FEEDBACK DRIVEN** ***PROCESS*** CONTROLLED BY **HUMAN PERCEPTION**

  UNLIKE THE ***MECHANISTIC, SELF REGULATING*** PROCESSES THAT DRIVE & CONTROL DEVELOPMENT & EVOLUTION OF BIOLOGICAL ORGANISMS TO YIELD ***STATISTICAL*** ADAPTATION

- DEVELOPMENT/ EVOLUTION **PROCESS** DETERMINES PROGRAM CHARACTERISTICS AND QUALITY

PRODUCT

⇑

PROCESS

- ***PROCESS*** THE KEY TO SATISFACTORY EXPLOITATION OF COMPUTER TECHNOLOGY

- ***DESIGN***, ***SUPPORT***, ***CONTROL*** RESPONSIBILITY OF ***SOFTWARE ENGINEERS*** = PROCESS ENGINEERS

SOFTWARE ENGINEER & PROGRAMMER

- TERMS INCREASINGLY USED SYNONYMOUSLY

- CONCEPTUALLY WRONG

- COUNTER-PRODUCTIVE

- ROLES ARE COMPLEMENTARY AND MUTUALLY SUPPORTIVE

PROGRAMMER

- PRIMARY TASK: STEP BY STEP **TRANSFORMATION** OF *APPLICATION CONCEPT* INTO *SOLUTION SYSTEM*

- EACH PROCESS STEP TRANSFORMATION OF A **SPECIFICATION** INTO CORRECT **IMPLEMENTATION**

```
-       CONCEPT VERBALISATION ⇒ REQUIREMENT
                  REQUIREMENT ⇒ SYSTEM SPECIFICATION
-               SPECIFICATION ⇒ EXECUTABLE REPRESENTATION
-               CHANGE NEEDED ⇒ IMPLEMENTATION
```

- OTHER **VITAL TASKS**
  - **SUPPORT** FOR USER
  - SYSTEM **MAINTENANCE**
  - SYSTEM **EVOLUTION**
  - **VALIDATION** OF EVERYTHING

- PROGRAMMING RELATES TO AND INCLUDES ALL INVOLVEMENT IN ANY ASPECT OF DEVELOPMENT OR EVOLUTION OF ***SPECIFIC***
  - SYSTEM ELEMENT(S)
  - or - PROGRAMS & SYSTEM(S)
  - or - FAMILIES OF SYSTEMS

PRODUCT ENGINEER

SOFTWARE ENGINEER

- PRIMARY CONCERN: ***DESIGN*, *CONTROL*, *SUPPORT*** DEVELOPMENT & EVOLUTION ***PROCESS***
    - **PROCESS ITSELF**
    - **METHODS**
    - **TOOLS**

- SELECT, DEVELOP & REDUCE TO PRACTICE
    - **METHODS**
    - **TECHNIQUES**
    - **PRACTICES**
    - **PROCEDURES**
    - **DIRECT TOOLS**
    - **GENERAL SUPPORT**

- ***INTEGRATE*** & INSTALL METHODS, TOOLS & IPSEs TO PROVIDE COHERENT ***PRODUCT*, *PROJECT* & *PROCESS*** SUPPORT FOR AN ***ORGANISATION*** & ITS ACTIVITIES OVER LIFE TIME OF EACH APPLICATION

- ***SOFTWARE ENGINEER CONCERN - PROCESSES BY WHICH SYSTEMS ARE DEVELOPED, PRODUCTS CREATED & MAINTAINED SATISFACTORY***

- INVOLVEMENT WITH SPECIFIC SYSTEM
    - **PROJECT DESIGN**
    - **PROCESS DESIGN**
    - **PLANNING**
    - **DEVELOPMENT OF PROJECT-SPECIFIC ⇒ METHODS**
      ⇒ TOOLS
    - **MANAGEMENT SUPPORT**
    - **PROCESSES MANAGEMENT**

PROCESS ENGINEER

TESTING IN A REUSE ENVIRONMENT
ISSUES AND APPROACHES

John C. Knight

Department of Computer Science
University of Virginia
Thornton Hall
Charlottesville, VA 22903

A Summary

Submitted To The Fourteenth Annual Software Engineering Workshop
Goddard Space Flight Center
Greenbelt, Maryland.

December 4, 1989

1. ISSUES

An economic advantage often claimed for reuse is that parts can be tested extensively before insertion into a reuse library. The term *certified part* is sometimes used to describe parts that have been tested prior to entry into a library (e.g., [11]) although certified is not a well defined term. There is the vague expectation that building software from tested parts will somehow make testing simpler or less resource intensive, and that products will be of higher quality [2, 6, 11]. For example, in [4] the potential productivity improvement through reuse is given for the entire lifecycle. The various aspects of testing are listed, and a potential reduction in cost resulting from reuse is shown for each.

Although using tested parts might offer some savings in testing, the situation is actually much more complex than this simple notion implies. The reuse paradigm raises many new issues in the area of testing, specifically:

(1) *Part quality.*
By definition, a part that is entered into a reuse library is being offered for use by others with the assumption that the more times it is used the better. This means that the part has to be prepared for *every* possible use if users are to have confidence that any phase of testing can be reduced or eliminated [10].

(2) *Distribution of parts.*
To maximize economic benefit a reuse library will be distributed widely, and parts will have to be built with portability in mind. They will also have to be tested so as to minimize the difficulties arising either from changes in the support environment or from porting.

(3) *Part adaption.*
Adaption, i.e., changing a part before it is used, is likely to be extensive with modern systematic reuse. Unfortunately, once a part is changed, the results of testing that took place prior to placing the part in the library cannot, in general, be trusted unless great care is exercised.

(4) *Adaptable parts.*
Adaption has been recognized as a necessity for generalized reuse to the extent that provision for it is finding its way into programming languages. Generic program units are present in Ada [12], for example, to support adaption and they present additional challenges for testing. The parameters used with Ada generic units are not merely for numeric or symbolic substitution. Subprograms can be used as parameters thereby allowing different instantiations to function entirely differently. This raises the question of exactly how, or even if, generic program units can be tested in any useful way [3].

(5) *Part use.*
A reusable part will be used in many different circumstances. Parts will contain assumptions about their use that may be undocumented yet must be complied with for correct operation. This indicates the need for increased attention being paid to integration testing during system development.

(6) *Part revision.*
Parts will be enhanced over time to improve their performance in some way yet maintain their existing interface. Systems built with such parts are then faced with a dilemma. Incorporating the revised parts might produce useful performance improvements but the resulting software will differ substantially from that which was originally built and tested.

(7) *Custom software.*
Although a new application might be built with parts from a reuse library, it will also

inevitably include custom software. The question that then arises is how to take advantage of the testing that has been performed on the parts to reduce the testing of the final system. Somehow test cases have to target the custom software rather than the reusable parts.

In summary, the various phases of testing that occur in a traditional development environment are still present but are changed in several ways when development is based on reuse.

2. APPROACHES

Modifications of existing techniques can be employed to deal with many of the issues raised above. Some issues, part revision for example, present problems that are similar to those which arise during maintenance. However, topics such as testing adaptable parts are not addressed by existing techniques. New approaches that address the problems of adaption and adaptable parts are discussed here.

Two forms of adaption are considered, *anticipated* and *unanticipated*. Anticipated adaption occurs when a user exploits facilities for change that were designed into the part, such as occurs with an Ada generic part or a part dependent on symbolic parameters. Unanticipated adaption occurs when a part is modified in a way that was not planned, usually using a text editor.

Anticipated Adaption

In many cases there are restrictions inherent in the design of a part to which any anticipated adaption must adhere. In the simplest case, a symbolic constant might be used to define a quantity such as the size of an array dimension. Adaption then consists of setting the symbolic constant prior to using the part, an action that was anticipated by the implementor of the part. The design of the part, however, might impose certain restrictions such as the size being within prescribed limits, or having some property such as being a power of two.

In a more general context, a functional restriction might be imposed on some piece of supplied program text. A procedure parameter to an Ada generic unit, for example, might be required to meet certain functional constraints inherent in the design of the generic unit.

In general, the checking that is required amounts to ensuring that an implementation (albeit often a small one) meets a set of specifications. Checking an anticipated adaption is, therefore, a special case of program verification in which the verification is of a source-to-source transformation. The restrictions correspond to the specifications and the adaption itself corresponds to the implementation. It is important to note that the specifications in this case do not derive from, and are not related directly to, the original specifications for the application. The specifications are a consequence of the design of the reusable part.

In a non-reuse setting, this verification will be performed by the author of a part. If the part is placed into a reuse library, however, the checks must be performed by the user. Correct use then relies on the restriction being documented correctly by the author, noticed by the user, and checked correctly by the user. Achieving correct use on a regular basis seems unlikely given this almost total reliance on human effort.

Specification languages like Anna [8] and Ada itself are not adequate to define the required checking. Our approach to dealing with anticipated adaption is to incorporate machine-processable statements of the required restrictions within the source text, and to check for compliance with restrictions after adaption but before traditional compilation. Such a notation cau be thought of as an assertion mechanism that is intended to operate at compile time rather than execution time.

This mechanism will not facilitate checking of restrictions such as required functionality. Using the analogy with program verification once again, we deal with adaptions that cannot be checked with a compile-time assertion mechanism using a testing system that again operates prior to conventional compilation. The concept is to associate a set of test cases with a part that must be executed satisfactorily by any code supplied as part of an adaption. The tests will be defined by the author of the part and executed by the user of the part.

Unanticipated Adaption

Arbitrary changes made using an editor are likely to be required frequently in attempting to reuse existing software. Such unanticipated adaption is far harder to deal with than anticipated adaption because its effect on the software is unpredictable. There is still the desire, however, to limit the amount of retesting that is needed since a part tailored specifically for reuse is likely to have been subjected to extensive unit testing to ensure part quality.

The problem that has to be dealt with in this case is precisely that of conventional program verification. Note, however, that the verification required is very different from the verification required with anticipated adaption. A modified part is different from the original part and obviously satisfies different specifications after unanticipated adaption.

Storing the specification of a part in machine-processable form and modifying the specification along with the part with extensive automated checking and support is the best way to deal with unanticipated adaption. Unfortunately, in general, this is not a practical approach to the problem. However, a promising first approach to dealing with many of the issues, at least partially, is the instrumentation of reusable parts with executable assertions [1, 8, 9]. In fact, Anna [8] is described as a notation for specification although it does not have the completeness characteristics of a rigorous approach such as VDM [5]. However, Anna does provide a rich notation for writing executable assertions.

The role of instrumentation using assertions is to include design information with the part, in particular to permit design assumptions to be documented in a machine-processable way. The effects of arbitrary changes cannot be checked with any degree of certainty in this way. However, there is some empirical evidence that executable assertions provide a useful degree of error detection when properly installed [7].

Adaptable Parts

As discussed above, the problem with anticipated adaption is to ensure that certain requirements imposed by the design of the part are met by the adaption. The problem of testing adaptable parts is the complement of this. It amounts to ensuring that the adaptable part will function correctly assuming that an adaption complies with the restrictions associated with design of the part.

The various adaptions that are provided with an adaptable part are similar in many ways to inputs to the part. From the point of view of correctness, setting a symbolic parameter, say, has some of the characteristics of reading an input of the same type as the parameter. The part should, in principle, operate correctly for every valid value of the parameter just as it should for every valid value of an input.

Adaptable parts cannot be executed without adaption. Each has to be given a ''value'' in order to use the part and the key question is whether the part will work correctly once these values are installed.

Our approach to testing of adaptable parts is based on a scenario in which the adaptable part is instantiated with specific values for the adaptions and then tested using some conventional approach to unit testing. Complete testing then consists of repeating this test process with systematic settings of the various adaptions. We are defining new coverage measures to assess the testing actually achieved.

REFERENCES

[1] Andrews, D.M. and J.P. Benson, ''An Automated Program Testing Methodology and Its Implementation'', *Proceedings of the Fifth International Conference on Software Engineering*, San Diego, CA, March 1981.

[2] Bassett, P.G., ''Frame-Based Software Engineering'', *IEEE Software*, July, 1987.

[3] Dowson, M, personal communication.

[4] Horowitz, E and J.B. Munson, ''An Expansive View of Reusable Software'', *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 5, September 1984.

[5] Jones, C.B., ''Systematic Software Development Using VDM'', *Prentice Hall International*, 1986.

[6] Lenz, M., H.A. Schmid, and P.F. Wolf, ''Software Reuse Through Building Blocks'', *IEEE Software*, July, 1987.

[7] Leveson, N.G., S.S. Cha, T.J. Shimeall, and J.C. Knight , ''The Use Of Self Checks And Voting In Software Error Detection: An'' Empirical Study" , submitted to *IEEE Transactions on Software Engineering*.

[8] Luckham, D.C. and F.W. von Henke, ''An Overview of Anna, a Specification Language For Ada'', *IEEE Computer*, March, 1985.

[9] Meyer, B., ''EIFFEL: Reusability and Reliability'', in *Software Reuse: Emerging Technology*, Tracz, W, (editor), IEEE Computer Society Press, 1988.

[10] Russell, G., ''Experiences Using A Reusable Data Structure Taxonomy'', Proceedings of the *Fifth Annual Joint Conference On Ada Technology and Washington Ada Symposium*, April 1987.

[11] Tracz, W., ''Software Reuse: Motivators and Inhibitors'', *Proceedings of COMPCON S'87*, 1987.

[12] U.S. Department of Defense, Ada Joint Program Office, *Reference Manual For The Ada Programming Language*, ANSI/MIL-STD-1815A, January, 1983.

VIEWGRAPH MATERIALS

FOR THE

J. C. KNIGHT PRESENTATION

TESTING IN A REUSE ENVIRONMENT
ISSUES AND APPROACHES

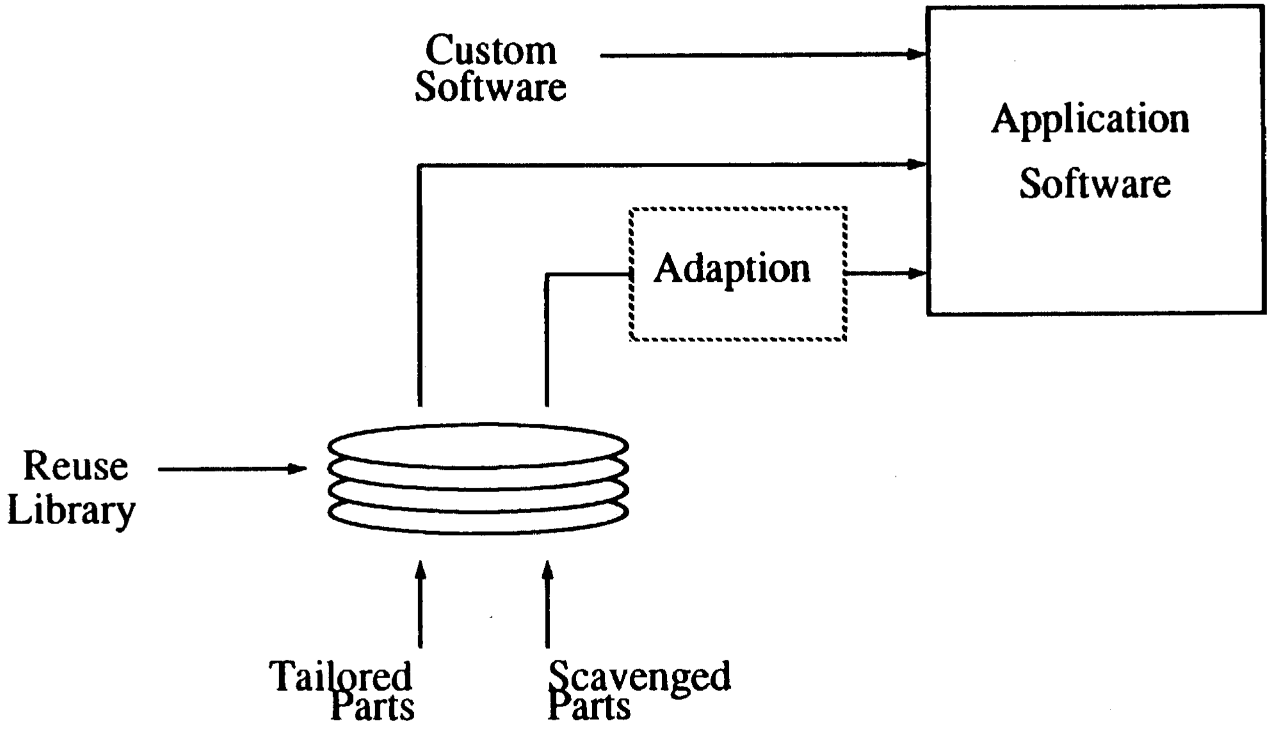
John C. Knight

Department of Computer Science
University of Virginia

and

Software Productivity Consortium

SOFTWARE DEVELOPMENT WITH REUSE



TESTING ISSUES WITH REUSE

- *Part Quality* – Is The Part Good Enough?
- *Part Distribution* – Will The Part Work If Moved?
- *Part Adaption* – Will The Part Work If Changed?
- *Adaptable Parts* – How Can We Test Parts Designed For Change?
- *Part Use* – Is The Part Being Used Properly?
- *Part Revision* – What If The Part Is Updated In The Library?
- *Custom Software* – How Do We Test The Custom Software?

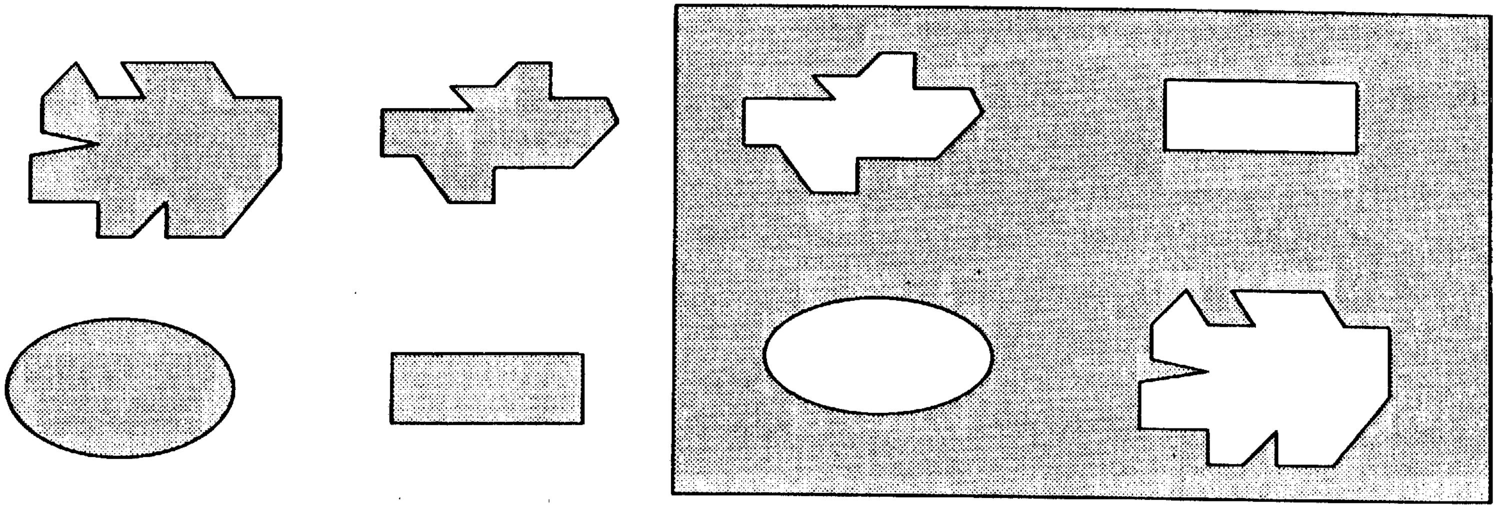
SOFTWARE PRODUCTIVITY CONSORTIUM

PART ADAPTION

- *Anticipated* Adaption:
  - Changes Planned By Part Developer
  - Setting Constants, Conditional Compilation, Generics
  - Problem Is Variant Of Traditional Verification
- *Unanticipated* Adaption:
  - Changes **Not** Planned By Part Developer
  - Ad Hoc Editing
  - Problem Is Traditional Verification

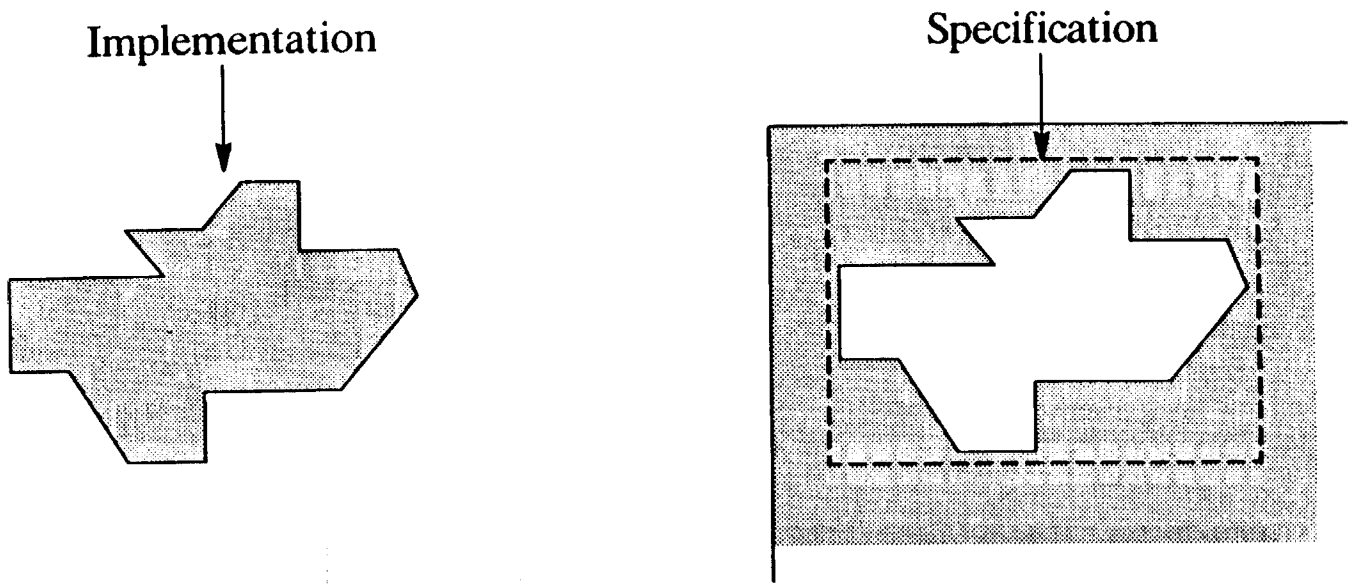
SOFTWARE PRODUCTIVITY CONSORTIUM

ANTICIPATED ADAPTION



- Examples:
  - Symbolic Parameter Must Lie Within Planned Range
  - Symbolic Parameter Must Be Power Of Two
  - Generic Procedure Parameter Must Provide Planned Semantics
- View Problem As Special Case Of Verification

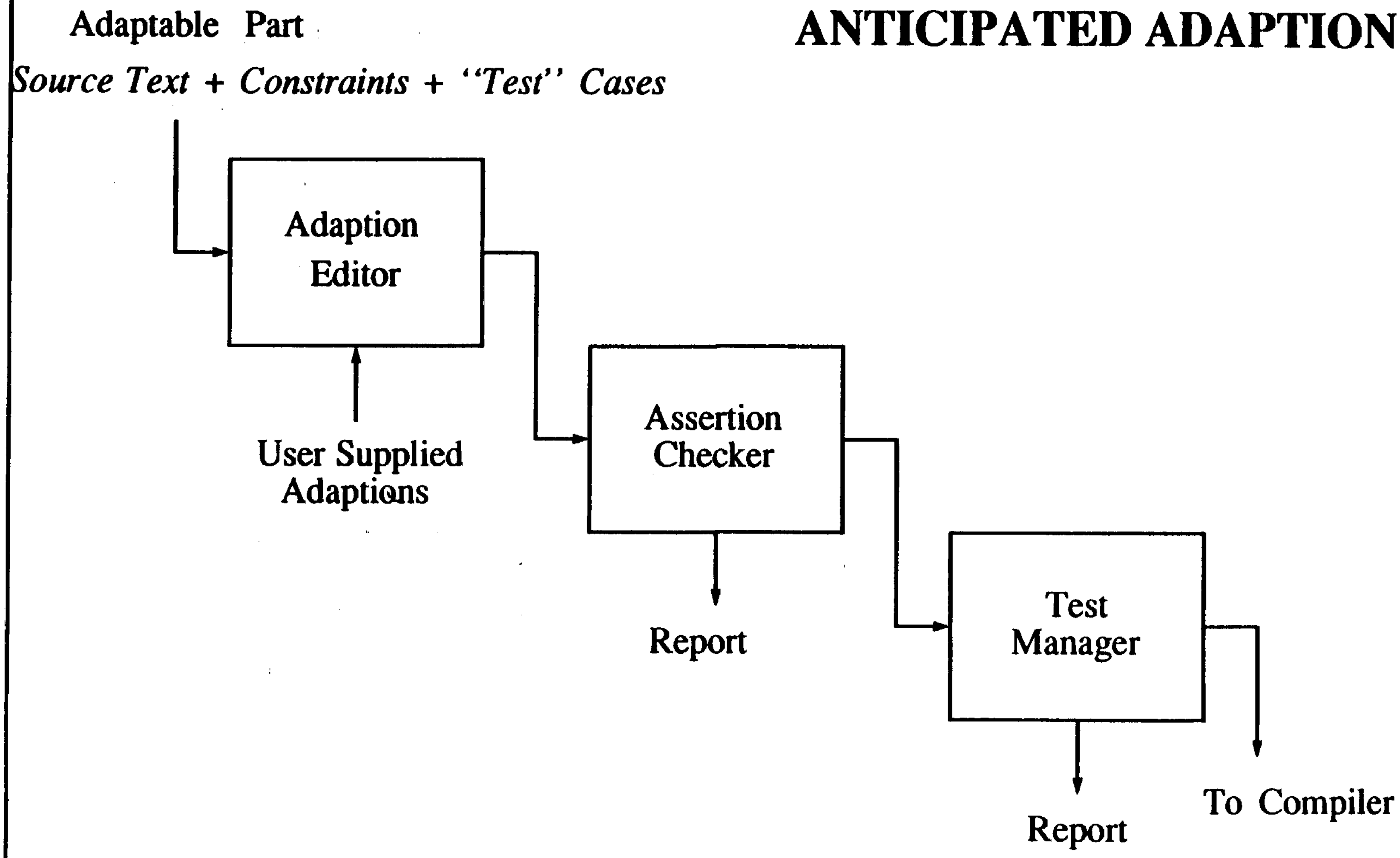
ANTICIPATED ADAPTION



- Specification Derives From Part Design, Not Part Functionality
- Verification Techniques:
  - "Proof" – Use Checkable Assertions
  - Testing
- Apply Both *After* Adaption, *Before* Compilation

SOFTWARE PRODUCTIVITY CONSORTIUM

CHECKING SYSTEM FOR ANTICIPATED ADAPTION



Adaptable Part

Source Text + Constraints + "Test" Cases

Adaption Editor

User Supplied Adaptions

Assertion Checker

Report

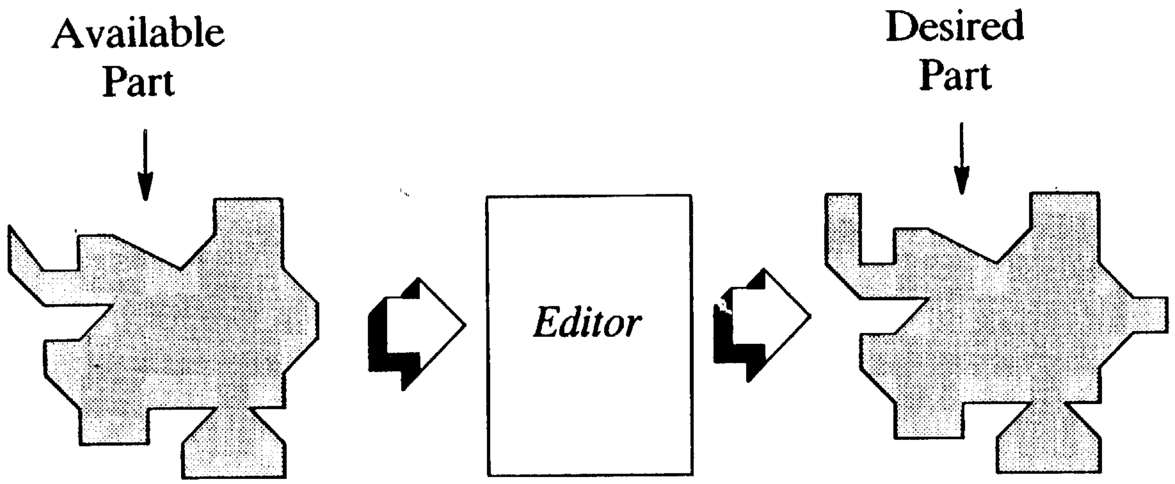
Test Manager

Report

To Compiler

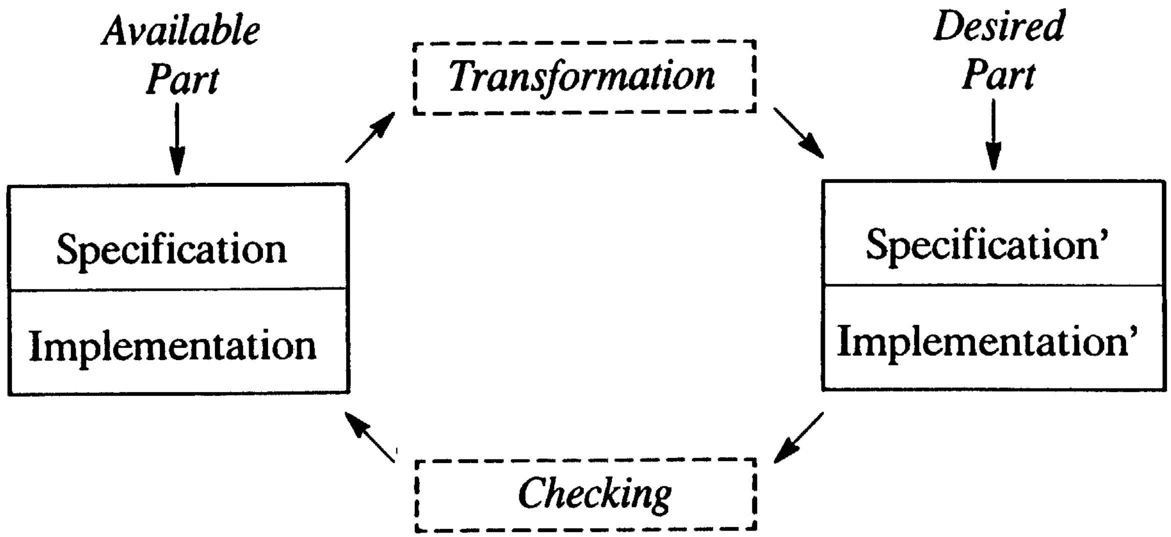
UVA
Department of Computer Science

UNANTICIPATED ADAPTION



- Unpredictable Change
- Part Specification Changes
- Problem Is “Pure” Verification:
  - Ensure Revised Specifications Are Met
  - Minimize Rework, Especially Testing

UNANTICIPATED ADAPTION



- First Approach:
  - Executable Specifications – ANNA
  - Coverage Measures
- Key Question – How Well Will This Work?
- Second Approach – Formal Specifications, e.g., VDM

ADAPTABLE PARTS

- Designed For Anticipated Adaption
- Changes Planned By Part Developer
- Use Support Facilities From Programming Language:
  - Setting Symbolic Constants
  - Conditional Compilation
  - Generics
- Setting Constants Can Be Complex, e.g., In Ada They Might Control:
  - Type Sizes, Structures, Representations, Available Space
  - Task Space, Priorities, Etc
  - Various Numeric Properties
- Part Has To Work Correctly For *Every* Use

ADAPTABLE PARTS

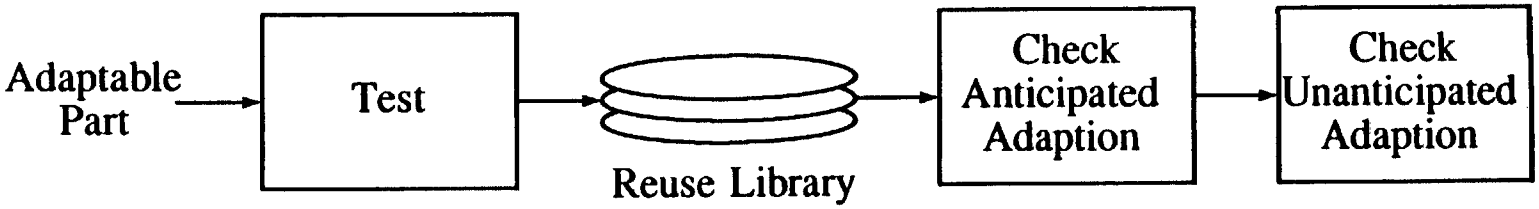
How Do You Test A Generic Part?

This Looks Really Tricky......

- The Other Half Of The Problem
- Adaptable Part Must Work With All Adaptions
- First Approach:
    - View Generic Parameters As “Inputs”
    - Develop Suitable Test Input Generators
    - Develop New Coverage Measures

SOFTWARE PRODUCTIVITY CONSORTIUM

SUMMARY
TESTING AND ADAPTION



CONCLUSIONS

- Reuse:
  - Affects The Way Testing Is Done
  - Does Not Necessarily Make Testing Easier
  - Gives New Opportunities For Faults To Arise
- Adaption Presents The Most Significant Challenges
- Preliminary Techniques Defined
- Preliminary Tools And Notations Defined
- Naive Economic Models Of Reuse Do Not Take Account Of The Impact Of Testing

Domain-Directed Reuse

Christine Braun
Rubén Prieto-Díaz
Contel Technology Center
15000 Conference Center Drive
Chantilly, VA 22021

Introduction

The Contel Technology Center's Software Reuse Project was established to introduce the practice of reuse throughout the corporation, with the objective of reduced cost and risk and improved quality in our software development efforts. We believe that the maximum benefits are achieved when reuse focuses on a particular application domain, making use of a standard design paradigm or architecture for that domain. This paper will explain this concept and describe our work in implementing such an approach.

Will reuse really make a difference?

It is currently popular for those considering the reuse problem to deplore the unwillingness of software developers to actually practice reuse, and to assert that, despite advances in supporting technology, little significant reuse occurs.

This is not true. Significant reuse (with significant savings) occurs:

- every time a real-time system is built on top of an existing operating system
- every time an information management system includes a Commercial-off-the-Shelf (COTS) DBMS
- every time a product vendor creates a new version of his product from parts of the old system
- every time a compiler builder "retargets" his compiler rather than building a new one from scratch
- every time a system designer draws on design knowledge from a previous similar system

What makes these cases of reuse successful? How can we learn from these successes and extend these benefits? Let us consider what they have in common. First, each *focuses on a particular application domain.* Each reuses large entities that perform domain-specific functions. Second, *each makes assumptions about the system architecture.* Systems can only make use of COTS operating systems, DBMSs, etc., if they are structured according to the model implemented by the COTS product. Product vendors must keep major architectures intact to allow reuse of existing parts when making upgrades. Compiler front-ends are reused because the overall compiler architecture is the same from one compiler to the next. Design is constrained to fit these architectural assumptions. Finally, *each is dependent on properly generalized and well-defined standard interfaces*. Many systems can use the same operating system or DBMS because the interfaces of these products are designed to accommodate a variety of needs and are well-understood and well-documented. Standard interfaces between compiler front-ends and back-ends allow reuse of these major components.

We believe that these successes can be extended by following the same model -- focusing on specific domains, developing standard architectures to direct and constrain designs toward the use of common components, and specifying standard interfaces to make reuse of these components possible. Contel's reuse project is taking such an approach, concentrating initially on the C^3I domain.

How can domain knowledge be incorporated in a reuse system?

Reuse researchers generally classify approaches to reuse as either *compositional* or *generative*. Compositional approaches support the bottom-up development of systems from a library of available lower-level components. Much work has been devoted to classification and retrieval technology, and to the development of retrieval systems to support this process. These systems are useful but do not meet our requirement for Domain-Directed Reuse.

Generative approaches are closer to what we are looking for. These are domain-specific; they adopt a standard domain architecture model and standard interfaces for the components. Their goal, however, is to *automatically* generate the new system from an appropriate specification of its parameters. Such systems can be immensely effective within particular narrow domains, but clearly their scope is limited. It is not realistic, at least with near-term technology, to imagine the completely automatic generation of real-time defense systems.

What *is* possible today is an approach that combines the two. We can draw on the domain analysis work that forms that basis for the generative approach, developing a standard domain architecture model and standard interfaces. We can then build an interactive system, as a "superstructure" on top of a general-purpose retrieval system like those that currently exist, that *directs* the designer through this architecture in the generation of the system. In effect, this is a generative approach that uses the human engineer as the generator, directing and constraining his choices to achieve the maximum reuse of available architecture components. Unlike the automatic generation approach, it allows human judgement and choice at each step, and recognizes the unlikelihood of developing the entire system from available parts. Its flexibility makes it applicable to most domains.

We envision a graphic user interface based on a representation of a standard domain architecture. This might, for example, provide a hierarchical breakdown of the architecture. The designer will initially be presented a top-level diagram showing the decomposition of the system into major subsystems. Pointing to the subsystem he wishes to work on, he will be given a display of the next-level decomposition, and so on. At any level in this process, the entity selected by the designer may be implemented by one or more components in the repository, and he will then step into the component selection process supported by the repository. Alternatively, he may ask for a further decomposition of the entity. For example, the designer of a C^3I system may initially select "Man-Machine Interface" as the area he wishes to design. The repository may have multiple Man-Machine Interfaces to offer the designer -- e.g. a fill-in-the-blanks forms interface and a menu-and-mouse interface. The user may select one of these or, deciding that neither meets his needs, he may ask for a next-level decomposition of Man-Machine Interface. At that level, he may decide to use a windowing package available in the repository. At each step, the components offered will conform to a standard interface definition adopted for the domain architecture model. Stepping through the hierarchy in this way directs the overall structure of the design so that it makes the maximum reuse of available parts, at the same time allowing the designer the discretion to substitute his own code whenever appropriate.

What have we done so far?

Our approach to developing a domain-directed C^3I reuse system has two threads -- *domain analysis* and *retrieval system development.* The domain analysis activity is focused on analyzing Contel's various ongoing C^3I programs and identifying future business and technical directions, to identify common structures, functions, and areas of potential standardization. In this work, we interact heavily with the company's C^3I "domain experts". We have currently completed a top-level domain study surveying the C^3I area at Contel and setting forth our objectives for the more detailed domain analysis. A full analysis, resulting in a generic system architecture, interface definitions, and a component classification scheme, will be completed early next year.

Because C^3I is only of the domains we wish to support, we will develop the C^3I retrieval system by building a domain-specific superstructure on top of a general baseline system. (In other words, we want to maximize reuse in building our reuse systems!) We have currently completed an initial baseline retrieval system; a second increment with a much improved user interface will be completed this year. Next year we will design and prototype the domain-specific user interface, based on the results of the domain analysis activity, and will work with the C^3I business groups to develop the necessary "building block" components to stock the repository. Contel's C^3I projects are already working with our initial system; they will adopt these new products as they become available and provide feedback that allows us to continue to improve our reuse capability.

What benefits can be expected?

Perhaps the best example of a field in which such a domain-based reuse model is already applied is compiler development. The existence of standard architectures and interfaces has made reuse the accepted practice. One Ada compiler vendor estimated that his organization had produced over 5 million lines of code in distinct compiler and tool products with approximately 100 man-years of labor -- a productivity rate of 50,000 lines of code per man-year. This is 10-20 times the usual programmer productivity estimate; clearly it was achieved because most of the software in later products was reused from earlier ones.

Such gains will not occur overnight in other fields; reuse via standard architectures has been accepted practice in the compiler field since its infancy. However, we believe that similar models are possible in other fields, and that comparable benefits can be achieved. The compiler field does not need a directed design tool to encourage and enforce use of standard architectures; no one would think of doing otherwise. However, in other fields, the hardest job will be changing, disciplining, and constraining ongoing design practices. The system we envision will do this easily and conveniently, improving productivity from the start and thus overcoming user resistance.

VIEWGRAPH MATERIALS

FOR THE

C. BRAUN PRESENTATION

CONTEL® Technology Center

DOMAIN-DIRECTED REUSE

Chris Braun

Rubén Prieto-Díaz

29 November 1989

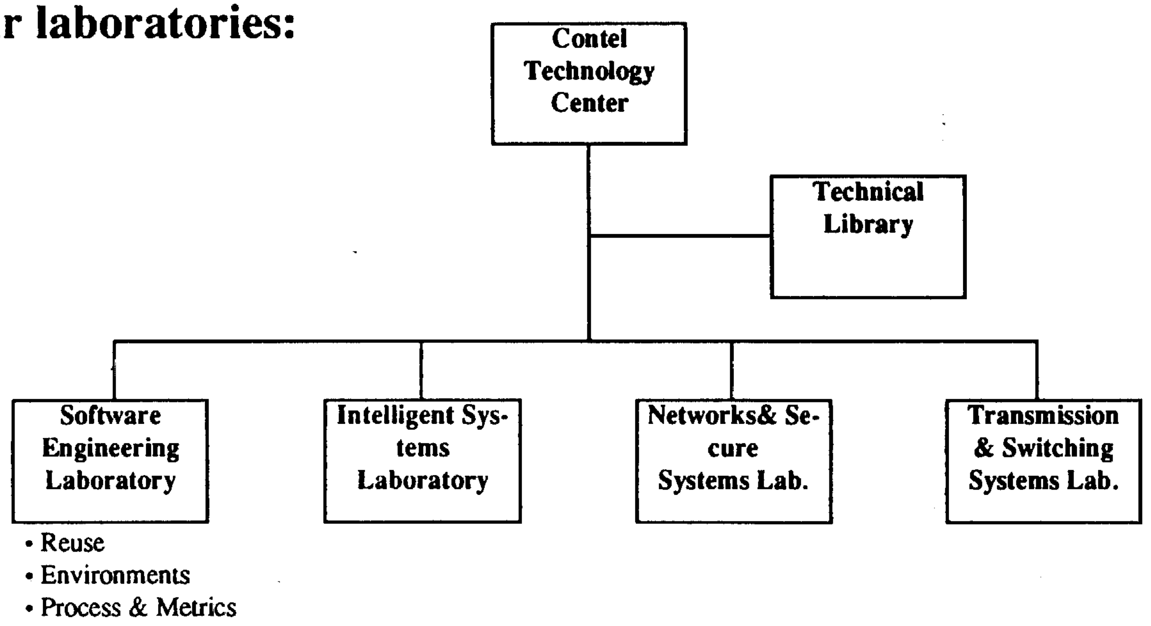
CONTEL Technology Center

THE CONTEL TECHNOLOGY CENTER

- corporate research group -- provides advanced technologies and related support to the divisions

- founded in 1988

- four laboratories:



Contel Technology Center

Technical Library

Software Engineering Laboratory
- Reuse
- Environments
- Process & Metrics

Intelligent Sys-tems Laboratory

Networks& Se-cure Systems Lab.

Transmission & Switching Systems Lab.

REUSE PROJECT GOAL

Conduct research in effective approaches to supporting software reuse, develop prototype reuse technology, and transfer that technology to Contel's business units.

WILL REUSE REALLY MAKE A DIFFERENCE?

- Is reuse real, or is it a lot of hype?
    - Yes, it's real, it happens a lot today, and people are really saving money.

- Oh yeah -- when?
    - using existing operating systems
    - using DBMSs
    - building a new system by modifying the previous version
    - retargeting a compiler
    - reusing design knowledge from an earlier project

WHAT DO THESE SUCCESSES HAVE IN COMMON?

- Each focuses on a particular application domain.
  - large, domain-specific entities are reused
- Each makes assumptions about the system architecture.
  - design choices are constrained to fit architecture
- Each is dependent on properly generalized and well-defined interfaces.
  - components are designed to accommodate a variety of needs
  - interfaces are well-understood and well-documented

PREMISE

- These successes can be extended by following the same model:
  - focus on specific domains
  - develop standard architectures to direct and constrain designs
  - specify standard interfaces to make reuse of these components

USING DOMAIN KNOWLEDGE

- *Compositional Reuse*
  - bottom-up construction from library of lower level components
  - supported by retrieval systems, classification schemes, etc.
  - => useful, but don't meet requirements
- *Generative Reuse*
  - based on standard domain architecture and interfaces
  - automatically generate system from specification of its parameters
  - => very effective in narrow domains, but limited in scope
- Our approach -- combine the two => DOMAIN-DIRECTED REUSE.

DOMAIN-DIRECTED REUSE

- Develop generic architecture and standard interfaces.

- Identify reusable components and catalog in a retrieval system.

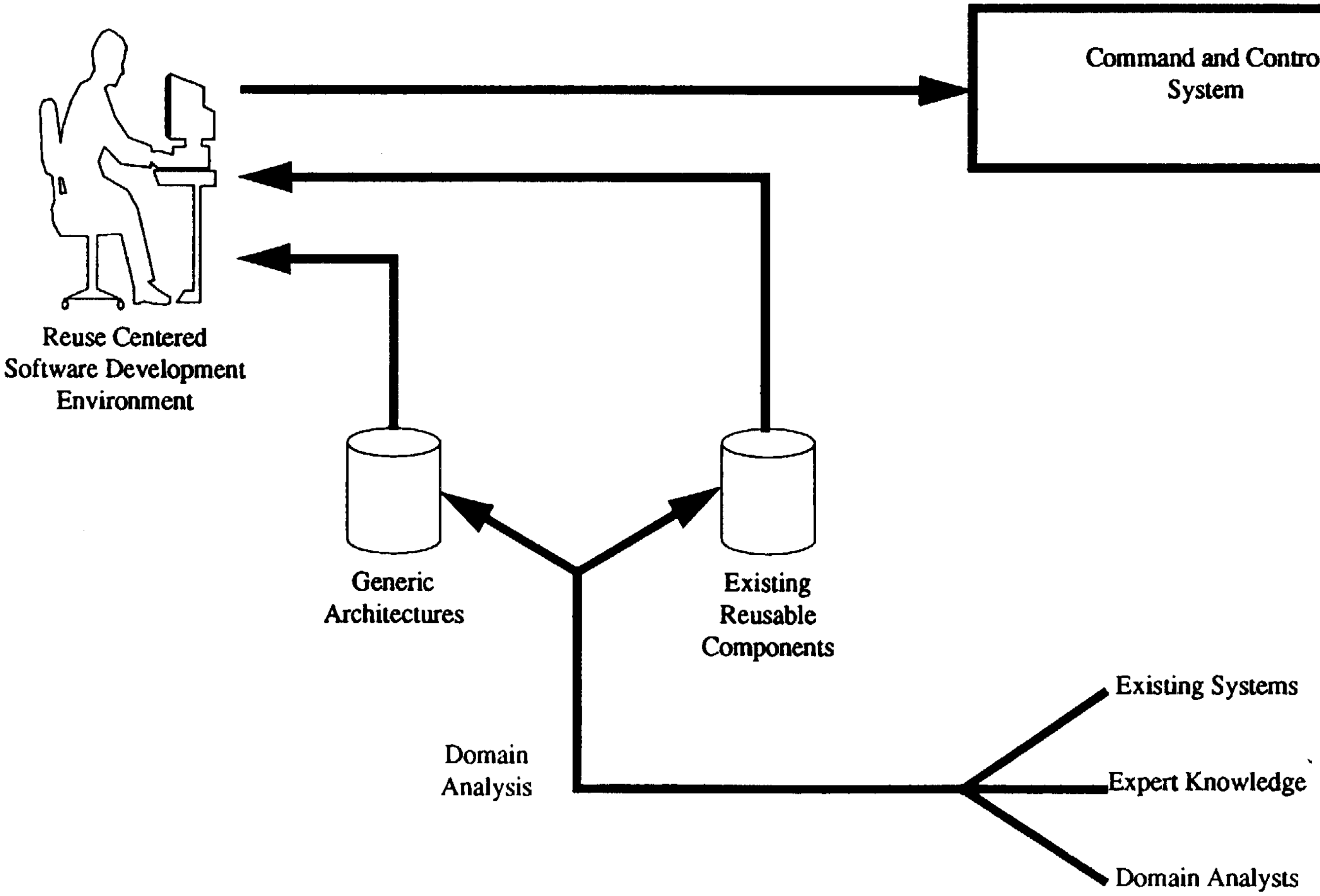
- Build interactive, domain-specific "superstructure" on top of the retrieval system.

  - directs the designer through the architecture, offering choice of components at each step (as available)

=> a generative approach using human engineer as the generator

  - allows human judgement and choice at each step
  - recognizes unlikelihood of developing entire system from available parts
  - flexible; suitable for most domains

A DOMAIN-DIRECTED REUSE ENVIRONMENT


Command and Control System
Reuse Centered Software Development Environment
Generic Architectures
Existing Reusable Components
Domain Analysis
Existing Systems
Expert Knowledge
Domain Analysts

HOW DOES THIS WORK?

- User sees graphical representation of top-level generic architecture -- e.g. for C^3I.

- Selects desired subsystem, e.g. Message Handling.

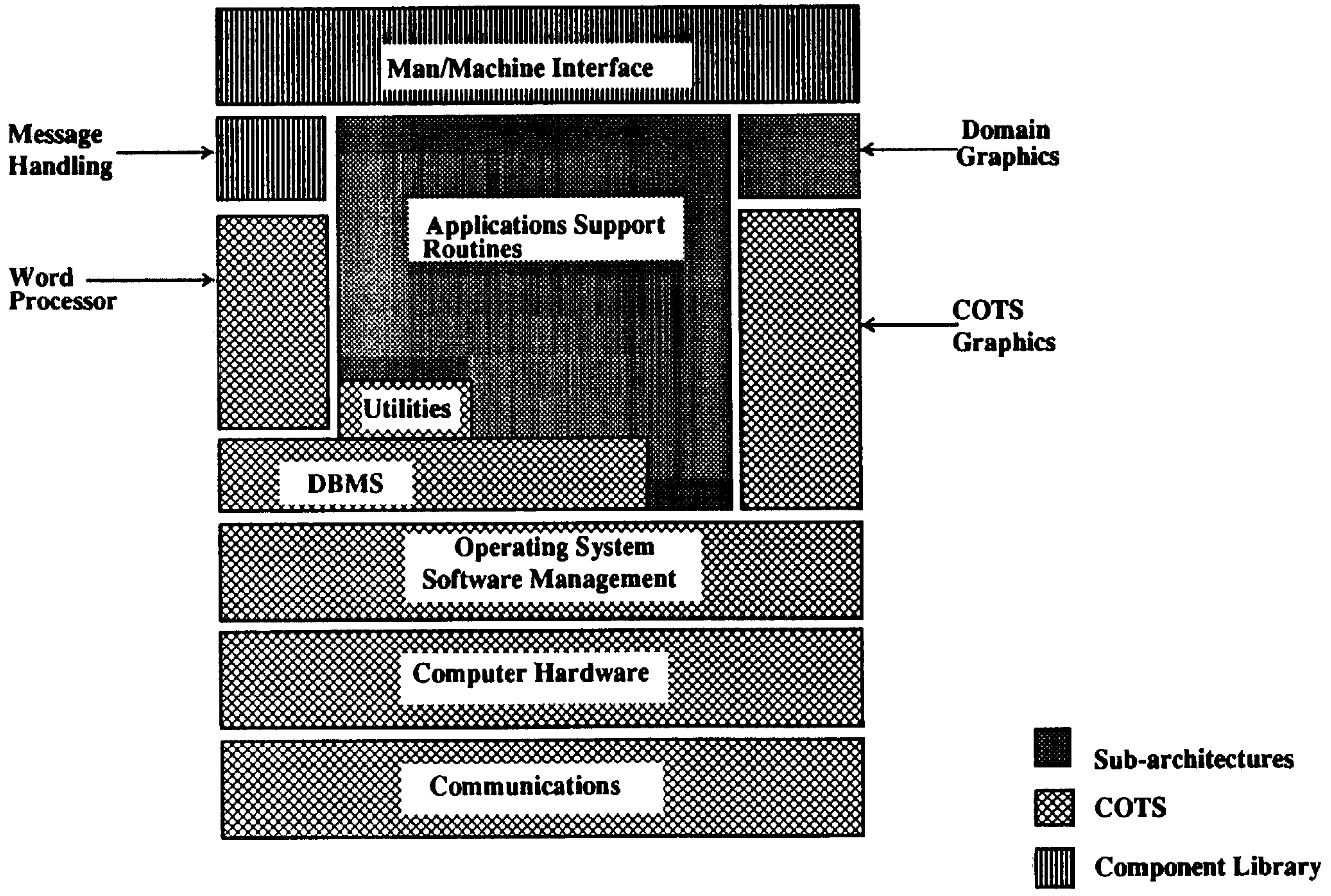
- Sees next-level decomposition.

- and so on

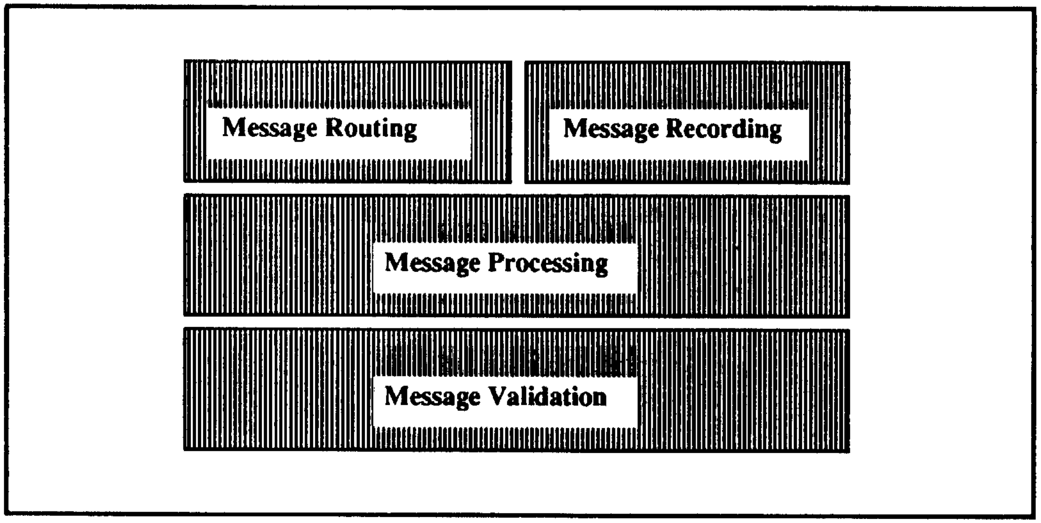
- At any level, can step into repository selection process to seek components that implement the function.

- Depending on results, can
  - select a component for use
  - decompose further
  - supply the component himself

A GENERIC C^3I ARCHITECTURE



MESSAGE HANDLING FUNCTIONS -- DECOMPOSED



WHERE ARE WE?

- C^3I domain analysis is ongoing.
  - studying current Contel systems, business directions, and developing technology
  - identifying common structures, functions, and areas of potential standardization
  - have defined initial top-level architecture
  - extensive interaction with company's C^3I "domain experts"
- Baseline retrieval system exists.
  - faceted classification scheme
  - window-oriented interface
  - basis for support for future domains as well
- Next year, we will:
  - design and prototype the domain-specific interface
  - work with C^3I projects to develop the needed "building blocks" to stock repository

WHAT BENEFITS CAN BE EXPECTED?

- Statistics from a compiler vendor:
  - 5 million lines of code in distinct compiler and tool products
  - 100 man-years
  - 50,000 lines/year -- roughly 10-20 times industry average
- How? REUSE!
- Can we do the same?
  - It won't come overnight; compiler people have been doing this forever.
  - They don't need tools to enforce this approach -- it's built in.
  - We do. It's hard to change.
  - We need to change, discipline, and constrain ongoing design practices. The envisioned system will "hold our hands" as we do so.

USING REVERSE ENGINEERING AND HYPERTEXT TO DOCUMENT AN Ada LANGUAGE SYSTEM

Presentation to

NASA Software Engineering Workshop

November 29, 1989

by Kent Thackrey

Planning Analysis Corporation
1010 N. Glebe Rd. Suite 890
Arlington, VA   22201
(703)-276-1250

THE PROBLEM

The Planning Analysis Corporation Ada Group Enterprise (PACAGE) provides a comprehensive set of Ada language services. One of our DOD clients had a personnel/manpower information system (an administrative system as opposed to a real-time or embedded system) which was written in EDL language for the IBM Series 1 computer. The client converted the system to Ada by translating instruction by instruction. This means that there was no Ada system engineering. This also means that there was no design or program documentation for the converted system, other than the code listings.

The new Ada system, on an IBM compatible PC, consisted of about 650 modules; a module is generally a function or procedure. There were approximately 40,000 lines of Ada code; each carriage return counting as a line.

PACAGE was asked to produce the design and program documentation in paper form. It was estimated that there would be about 2500 sheets of paper. Such documentation in paper form would have been laborious to produce, difficult to access, and probably would not be maintained.

THE SOLUTION

We recommended instead that PACAGE automate the documentation process as much as possible and produce on-line documentation instead of paper.

Our process consisted of two stages:

1. Reverse engineering the Ada code to produce documentation, using a partially automated process.

2. Developing an on-line documentation workbench using hypertext.

Hypertext

There are several ways to access information. If we read a book from front to back, for example, we access the information sequentially. The drawback with this method is that sometimes we have questions about the material that are answered later in the text and it is difficult to track down answers and still maintain continuity in what we are reading.

On-line information retrieval systems typically use a menu structure to organize the information systematically and to allow the user to retrieve it. To view information on a branch other than the current branch of the tree structure, we have to work back through the main menu. This also makes it difficult to maintain our train of thought if there is much information.

Hypertext organizes the information using a network system. Typically, a hypertext system is developed hierarchally, like the typical multi-layered menu system, but it allows the viewer to go directly from any node to any other node on any branch. A programmer investigating a module can use the documentation workbench, for example, to

- Read the module description
- Examine its screen layouts
- Read the module description of a calling routine or called routine
- View the source code
- Look up data elements in the data dictionary

Most of the transitions between these interdependent information sources are made with single keystrokes, following pre-established links. This allows us to access information following our normal thought patterns.

Other hypertext features allow the viewer to retrace his steps or return directly to the main menu, to mark a piece of documentation for future reference (viewing, saving, or printing), and to search the documentation for any string.

<u>The Process</u>

Figure 1 shows what information was included in our documentation workbench and how it was linked and accessed.

A four step process was used to reverse engineer the code to create the documentation:

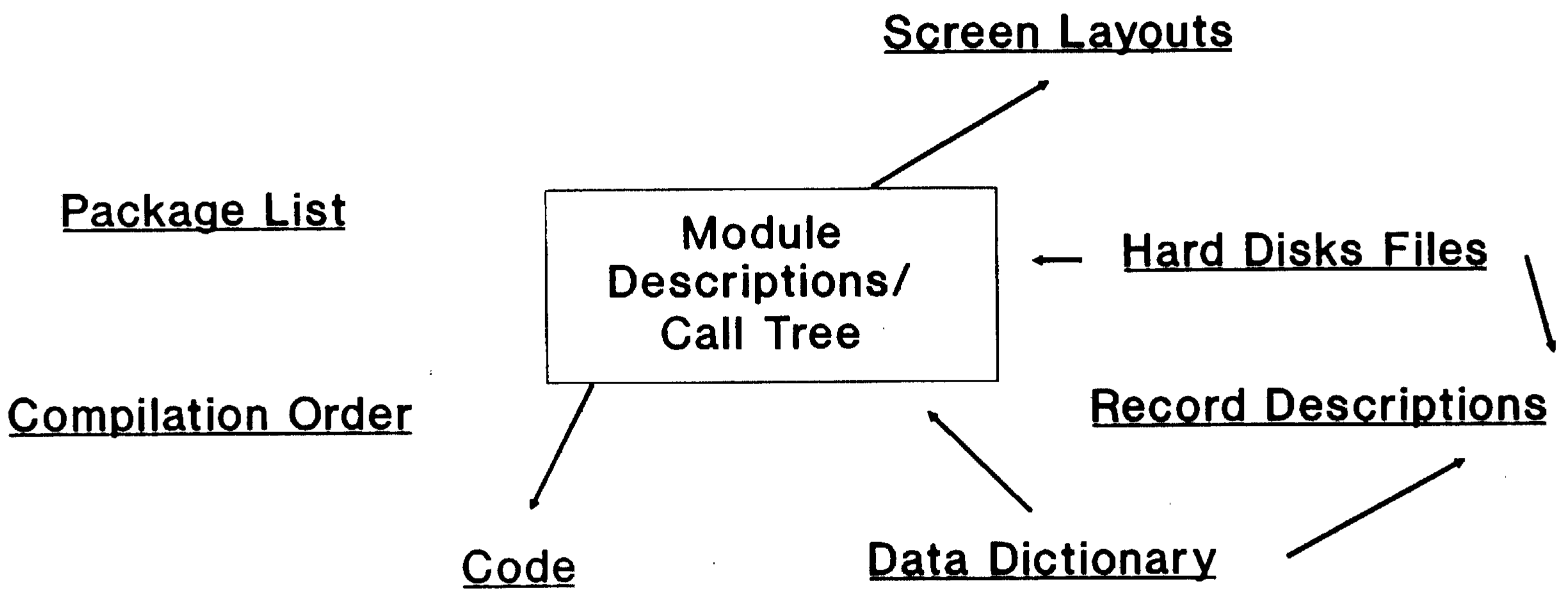
1. A call tree was created by manually examining the code. The call tree was then documented on-line using the Houdini hypertext software. The call tree shows graphically which modules call which other modules.

2. A code parsing program was written in PROLOG to examine the code and store information about the modules in a database. This information consisted of a list of which modules called and were called by a given module, the calling statement format, a description of the input and output parameters, the location of spec and body code, record and screen information, and other data.

3. A shell generation program was written in PROLOG to take the module information from the database and generate most of the on-line documentation within the Houdini hypertext call tree structure.

4. The source code was examined again manually to extract information that the code parser could not. This included information such as a purpose statement for each module and a

Ada DOCUMENTATION WORKBENCH



Additional Access:

- Thru menu to all except Code
- Thru Alphabetic Reference List to all except Package List and Compilation Order

FIGURE 1

description of the global variables that were used.

This information was pulled together in an easy to use, on-line workbench.

THE RESULTS

Approximately 2500 man hours were spent on this project. Of this, 25 percent was spent on analysis, 25 percent putting together the hypertext structure, 30 percent pulling all of the text together, and 20 percent on reviewing, testing, and implementing the workbench.

The overall results of the project have been very positive:

- The on-line documentation was accepted as meeting the requirements of the DOD client.

- There have been no requests to provide paper documentation since the workbench was implemented.

- Users report that the system is easy to use and, at least initially, there has been significant use.

In future versions we recommend that improvements be made to facilitate maintenance. Our client has made significant changes to the personnel/manpower information system without adequately maintaining the documentation workbench. As a result, usage has dropped recently. We recommend with future versions that:

- A list of called by and called from information be removed from the module descriptions because it is difficult to maintain and can be accessed through the call tree.

- The client receive more hypertext training so they are proficient at making changes. Although not a lot of expertise is required to maintain a hypertext system, there should be some.

FUTURE DIRECTIONS - THE IDEAL DOCUMENTATION WORKBENCH

Based on our experiences with this documentation workbench, we have developed some plans for future workbenches. The ideal documentation workbench would serve two purposes:

- To facilitate SYSTEM MAINTENANCE

- To facilitate REUSE

We make the assumption that the workbench will be used by people with average technical ability and with no prior knowledge of the application.

<u>Maintenance Documentation</u>

The purpose of the maintenance documentation included in the workbench is to:

- Make it easy for designer/analysts to design code changes properly.

- Make it easy for designer/analysts to assess the impact of a change on other parts of the system.

- Make it easy for programmer/analysts to quickly locate and fix bugs in the system.

Ideally, for new systems, this workbench will be developed as a by-product of the system development process with minimal additional effort. It will contain only that documentation that is necessary for maintenance of the system, and not contain documentation that is for development purposes only. Organizations vary on what documentation they consider necessary for maintenance, but to be included in the workbench it must be documentation that will be maintained as the system is maintained. This means that documentation changes should be a by-product of the normal maintenance effort with minimal additional effort.

Reuse Documentation

The purpose of the reuse documentation that we include in our documentation workbench is to provide information in a form and format that can be easily transported to a reuse library where:

- It will be available for other applications.

- Appropriate search mechanisms are available to effectively locate and investigate modules.

- Appropriate metric gathering mechanisms are available to track the usage of the modules.

- Code and documentation can easily be transported to new applications.

All modules that are part of our new application will be documented in our documentation workbench. A subset of these will be identified during the system development process as candidates for reuse and have additional documentation included for them in the workbench. As with the maintenance documentation, most reuse documentation should be generated as a by-product of the system development process with minimal addition effort.

Sample Workbench

Based on this criteria for maintenance and reuse documentation, we have designed a sample documentation workbench. Figure 2 shows how this sample workbench would be structured and linked together.

SAMPLE DOCUMENTATION WORKBENCH

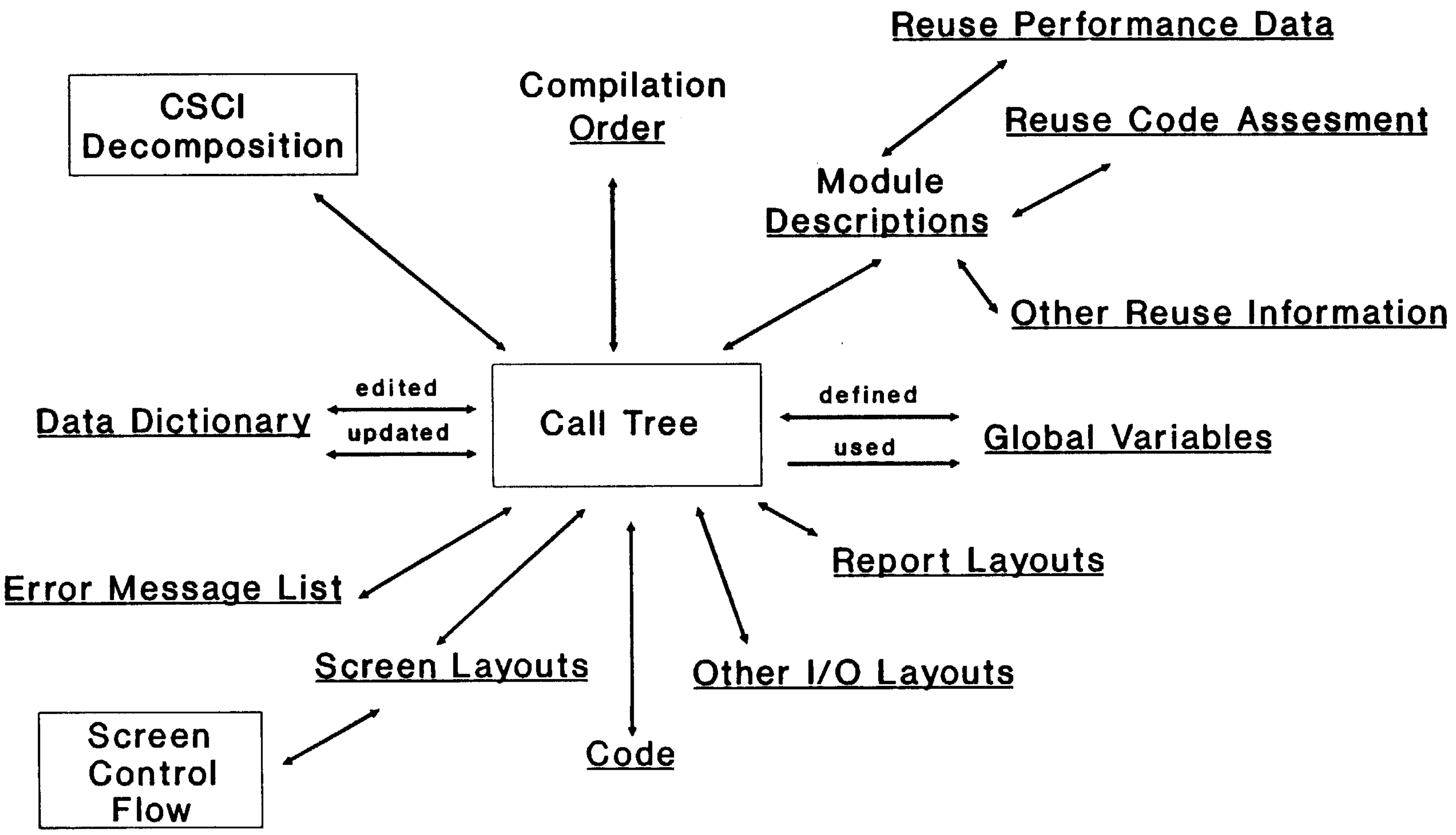


FIGURE 2

Each box represents a separate hypertext structure where there will be internal links between pieces of information in addition to the external links shown in the chart.

Good hypertext systems have a visible structure to them. It would be possible to establish many more links between the pieces of information shown in the chart, but a logical structure is important to avoid spaghetti hypertext, just like we avoid spaghetti code in computer programs. This makes it easy for the viewer to understand and for the technician to maintain. In this case, we are using the call tree as the focal point for all of the links.

This is a sample of the steps that would be involved in developing this workbench for an MIS type system. These steps would normally be embedded in the system development methodology and have been extracted here to illustrate the process:

1. Develop data dictionary.
2. Develop CSCI decomposition chart in hypertext.
3. Develop screen layouts, report layouts, and other I/O layouts.
4. Develop screen control flow in hypertext;
   establish links from screen control flow to screen layouts.
5. Develop call tree and module description shells in hypertext;
   establish links from call tree to CSCI chart, module description shells and to screen, report, and I/O layouts.
6. Develop reuse shells in hypertext for reuse candidates;
   establish links from reuse shells to module descriptions.
7. Fill module description information and reuse description information into the shells.
8. Develop global variable list, error message list, and package list;
   establish links from call tree to these and to the data dictionary.
9. Establish compilation order;
   establish link from compilation order to the call tree.
10. Develop code;
    establish link from code to call tree.
11. During informal testing, capture code analysis data and enter into reuse shell.
12. During stress testing, capture performance data and enter into reuse shell.
13. At implementation, port hypertext documentation for reusable modules to reusable library.

The documentation workbench, of course, would be customized for each organization. The steps necessary to create it would be customized and mapped onto the organization's system development methodology.

THE BENEFITS

There are several benefits to developing a hypertext documentation workbench similar to the sample here:

The system is easier to understand and maintain. A good hypertext structure makes it easier to navigate through the information in a logical manner. In addition, a workbench like the one in this example forces a structure on the code that will make the code easier to maintain. By linking the data dictionary element to the module in the call tree where it is edited, we are requiring that the element be edited in only one module. This is good programming practice and the workbench structure will enforce it. Similarly, we can force that a individual screen be generated in only one place, a generic error message be displayed from only one place, or a data element be updated from only one place. If we do this, then we can use the call tree to track down every place that the screen, error message, or data element update is generated.

Provides easily accessible reuse documentation. All of the modules that are candidates for reuse will be available for reuse on the application being developed through the documentation workbench. Later they will be available for reuse on other applications through the reuse library. The hypertext links will make it easier to examine all of the reuse documentation associated with a module. Also, hypertext has a mark and save feature and a memory residence feature. Conceivably, a programmer in a text editor could press a hot key to activate the hypertext memory residence feature and thereby enter the reuse library. After examining the reuse documentation and selecting a code module, the programmer could use the mark and save feature to save the selected piece of code to a file and then import it into the text editor for use on the current application.

Facilitates reuse analysis. The hypertext structure in the documentation workbench provides some guidelines to use when deciding if a code module is a reuse candidate. For example, if the code module has links to a screen layout, a report layout, or to a global variable, it probably is not a good candidate. If there is a link from a data element to an edit routine, that routine may not be a good candidate, but the routine it calls may be. For example, a routine to validate a termination date probably has some application-specific edit checks, but the generic date routine it calls probably is a good reuse candidate.

Improves the development process. In addition to forcing a useful structure on the code, we have also forced a structure on the development process. For example, hypertext provides built-in traceability. If it is not clear where to establish a link or if there is no place to tie down a link, then we probably need to make improvements to the system. Hypertext has been used, for example, to link a system design to its requirements.

Provides more easily maintainable documentation. Most of the documentation in the workbench is:

1) Dynamic in the sense that when it is changed the workbench is automatically updated. By linking to the actual source

code, for example, a code change is automatically reflected in the workbench.

or

2) Redline documentation in that the designer/analyst would normally mark it to communicate to the programmer what changes are to be made. It will take minimal effort to change an on-line screen layout or report layout from a redlined copy.

or

3) Producible through an automated reverse engineering process. It would be straightforward to write a code parser to generate the call tree from the Ada code. The global variable list and most of the module description information could also be generated from a code parsing program. If significant changes are made much of the revised documentation could be generated automatically.

SUMMARY

Hypertext is deceptive in that the concept is very simple, but its uses are many and its impact can be significant. Each hypertext application tends to generate ideas for bigger and better uses the next time. Some work is being done now using hypertext to develop system documentation and reuse documentation, but our industry is just getting its feet wet with hypertext compared to where it will be in the not too distant future.

VIEWGRAPH MATERIALS

FOR THE

K. THACKREY PRESENTATION

USING REVERSE ENGINEERING
AND HYPERTEXT TO
DOCUMENT AN Ada LANUGUAGE SYSTEM

Presentation to

NASA Software Engineering Workshop

November 29, 1989

by Kent Thackrey
Planning Analysis Corporation

Ada DOCUMENTATION WORKBENCH

The Problem

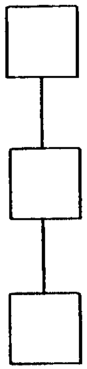
- 650 undocumented Ada modules (40,000 LOC)

- MIS type system on a PC

- Documentation would require 2500 sheets of paper
  - Laborious to produce
  - Difficult to access
  - Probably would not be maintained

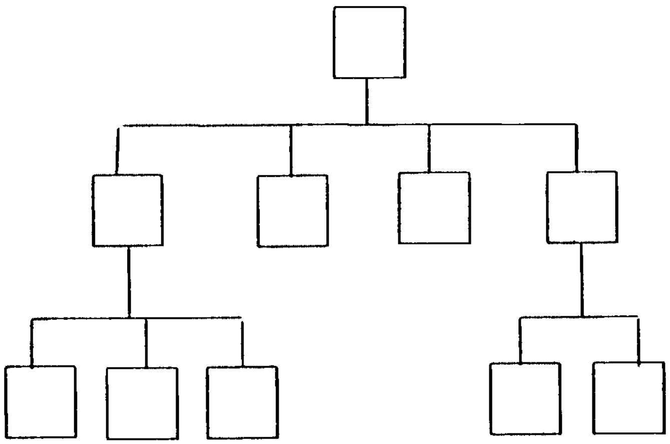
DEFINITION OF HYPERTEXT

Sequential Access



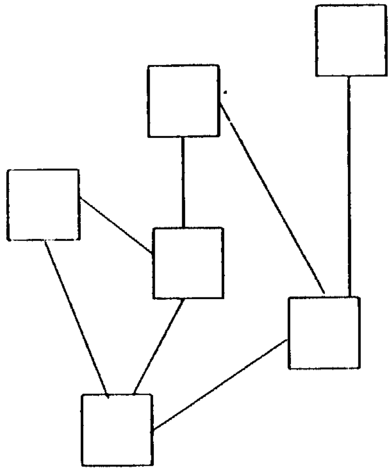
Example:
Pages of a
Book

Hierarchical Access



Example:
Menu Access
Computer System

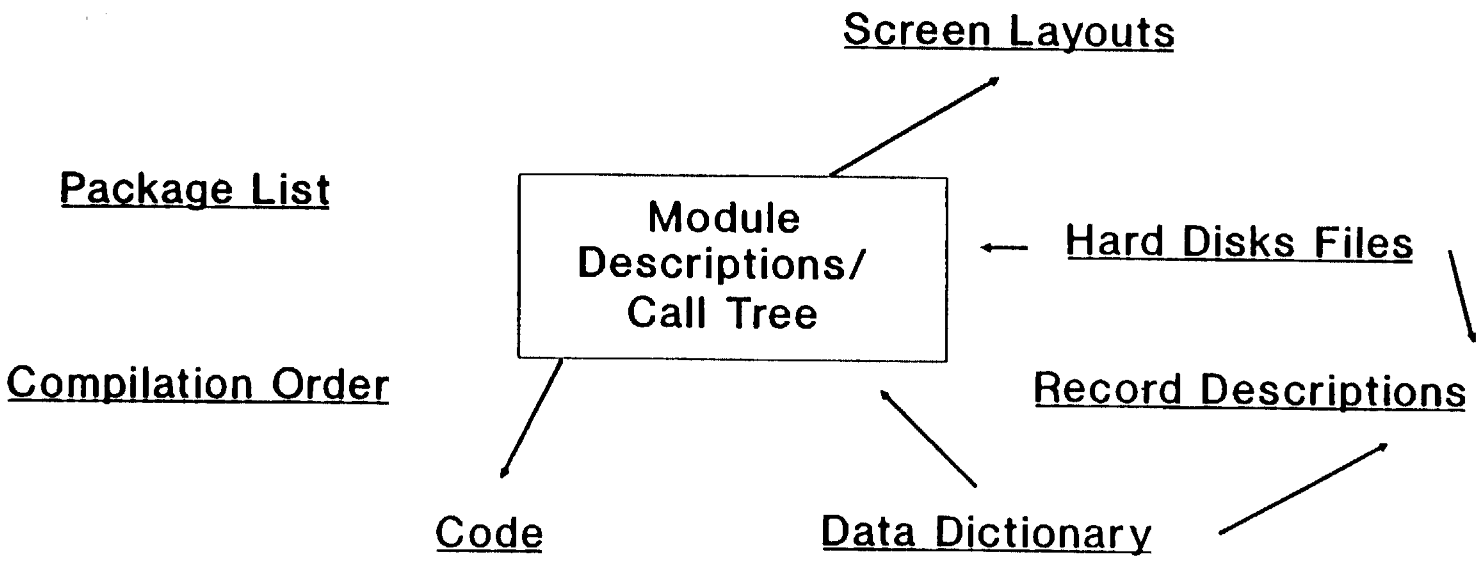
Hypertext Access



Example:
Anything that can
be in ASCII File

PLANNING ANALYSIS CORPORATION

Ada DOCUMENTATION WORKBENCH



Screen Layouts

Package List

Module Descriptions/ Call Tree

Hard Disks Files

Compilation Order

Record Descriptions

Code

Data Dictionary

Additional Access:

- Thru menu to all except Code
- Thru Alphabetic Reference List to all except Package List and Compilation Order

Ada DOCUMENTATION WORKBENCH

The Process

1. Hypertext call tree developed
   - Manually examine code
   - Houdini to organize Hypertext links

2. Ada code parsing program in Prolog

3. Shell generation program in Prolog

4. Manually extract remaining info and complete links

Ada DOCUMENTATION WORKBENCH

Results

- 2500 Man Hours
  - 25% Analysis
  - 25% Develop Structure
  - 30% Pulling Text Together
  - 20% Review, Test, and Implement
- Positive Results
  - No Requests for Paper
  - Acceptable to DoD Client
  - Significant use Initially
- Recommended Improvements to Facilitate Maintenance
  - Remove Call Tree Info From Module Description
  - Organize Links Better
  - Provide Basic Hypertext Training

IDEAL DOCUMENTATION WORKBENCH

PURPOSE:

- To facilitate MAINTENANCE
- To facilitate REUSE

AUDIENCE:

- People with average technical ability
  and no knowledge of the application

MAINTENANCE DOCUMENTATION

PURPOSE:

- Easily design changes
- Easily assess impact of changes
- Easily locate and fix bugs

FEATURES:

- Not contain development-only documentation (or documentation that won't be maintained)
- By-product of normal development effort
- Documentation changes by-product of normal maintenance effort

PLANNING ANALYSIS CORPORATION

REUSE DOCUMENTATION

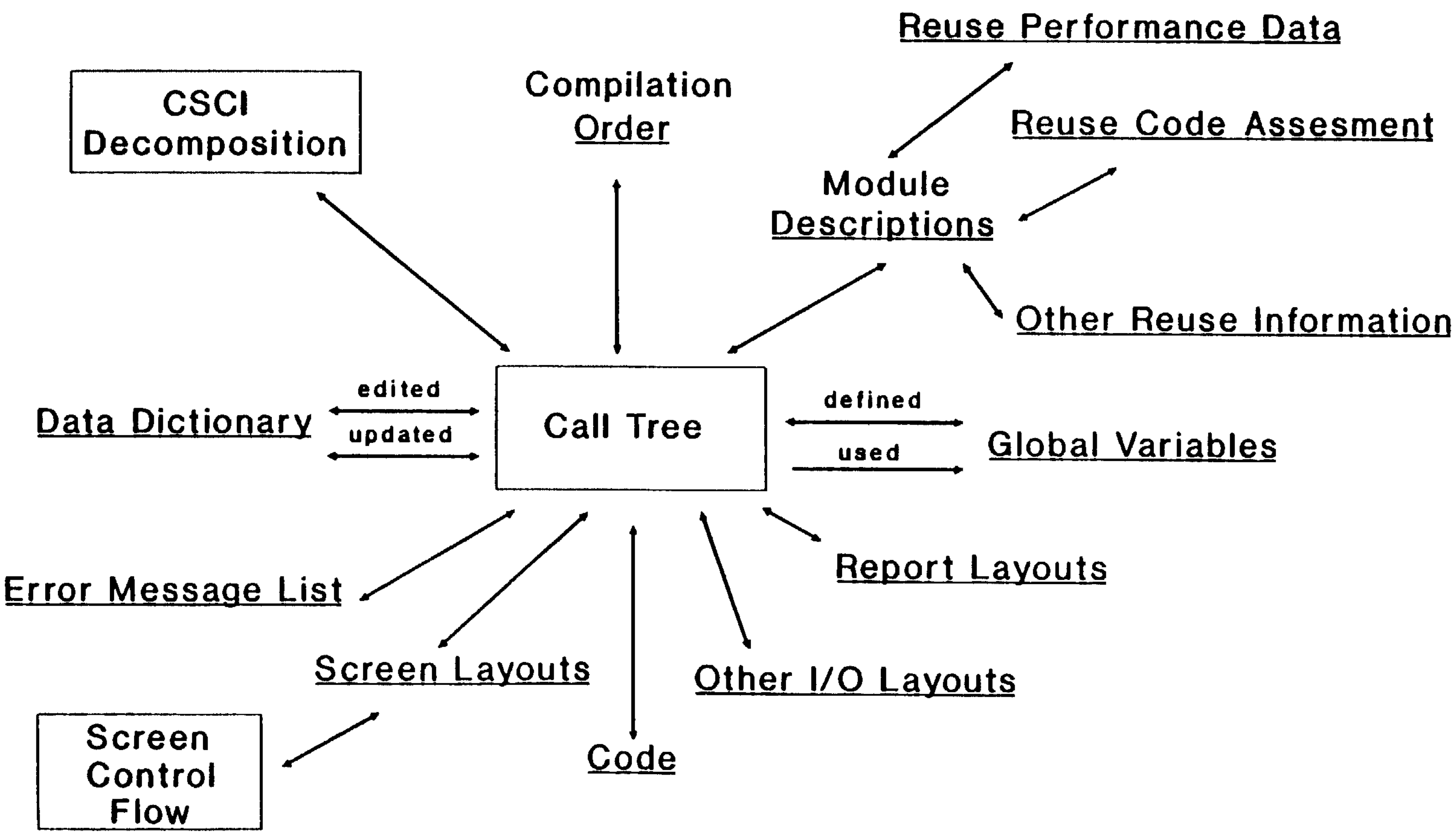
PURPOSE:

- To be in a Format and Form to be Easily Transportable to a Reuse Library
  - Available for Other Applications
  - Appropriate Search Mechanisms Available
  - Appropriate Metric Gathering Mechanisms Available
  - Code and Documentaion Easily Transported to New Applications

FEATURES:

- Extension of Maintenance Documentation
- Modules Identified as Reusable During Development Process
- Generated as By-Product of Development Process

SAMPLE DOCUMENTATION WORKBENCH



PLANNING ANALYSIS CORPORATION

SOFTWARE DEVELOPMENT PROCESS
(Example for MIS Type System)

1. Data Dictionary

2. CSCI Decompositon in Hypertext

3. Screen Layouts, Report Layouts, other I/O Layouts

4. Screen Control Flow in Hypertext; Links to Screen Layouts

5. Develop Call Tree and Module Description Shell in Hypertext
   and Define Modules; Links to CSCI chart and Screen,
   Report, and I/O Layouts

6. Reuse Shells for Reuse Candidates in Hypertext;
   Links to Call Tree

7. Module Descriptions and Reuse Descriptions

SOFTWARE DEVELOPMENT PROCESS (Continued)

(Example For MIS Type System)

8. Global Variable List, Error Message List, Package List; Links from these and Data Dictionary to Call Tree

9. Compilation Order; Link to Call Tree

10. Develop Code; Links to Call Tree

11. During Informal Testing, Capture Code Analysis Data into Reuse Shell

12. During Stress Testing, Capture Performance Data into Reuse Shell

13. At Implementation, Port Hypertext Documentation for Reusable Modules to Reuseable Library

BENEFITS

- System Easier to Understand and Maintain
- Provide Easily Accessible Reuse Documentation
- Facilitates Reuse Analysis
- Improved Development Process
  - Forced Additional Structure
  - Built-in Traceability
- More Easily Maintainable Documentation

PLANNING ANALYSIS CORPORATION

SESSION 4 — TESTING AND ERROR ANALYSIS

R. W. Selby, University of California, Irvine

M. Bush, JPL

M. Hecht, SoHaR, Inc.

Classification Tree Analysis Using the *Amadeus* Measurement and Empirical Analysis System

Richard W. Selby,
Greg James,
Kent Madsen,
Joan Mahoney,
Adam A. Porter, and
Douglas C. Schmidt

Department of Information and Computer Science[1]
University of California
Irvine, California 92717
(714) 856-6326
selby@ics.uci.edu

[1] This work was supported in part by the National Science Foundation under grant CCR–8704311 with cooperation from the Defense Advanced Research Projects Agency under Arpa order 6108, program code 7T10; National Aeronautics and Space Administration under grant NSG–5123; National Science Foundation under grant DCR–8521398; University of California under the MICRO program; Hughes Aircraft Company; and TRW.

Abstract

Classification tree analysis is a metric-driven technique that categorizes software components according to their likelihood of having user-specified properties such as high error-proneness or high development cost. This paper outlines a method that uses classification trees as metric integration mechanisms, enabling the synergistic use of numerous metrics simultaneously. An extensive validation study has been conducted using NASA project data and another is underway using Hughes project data. This paper summarizes the empirical results from using classification trees as predictors of high-risk software components in these environments. Classification analysis, along with other empirically-based analysis techniques for large-scale software, will be supported in the *Amadeus* measurement and empirical analysis system.

1 Introduction

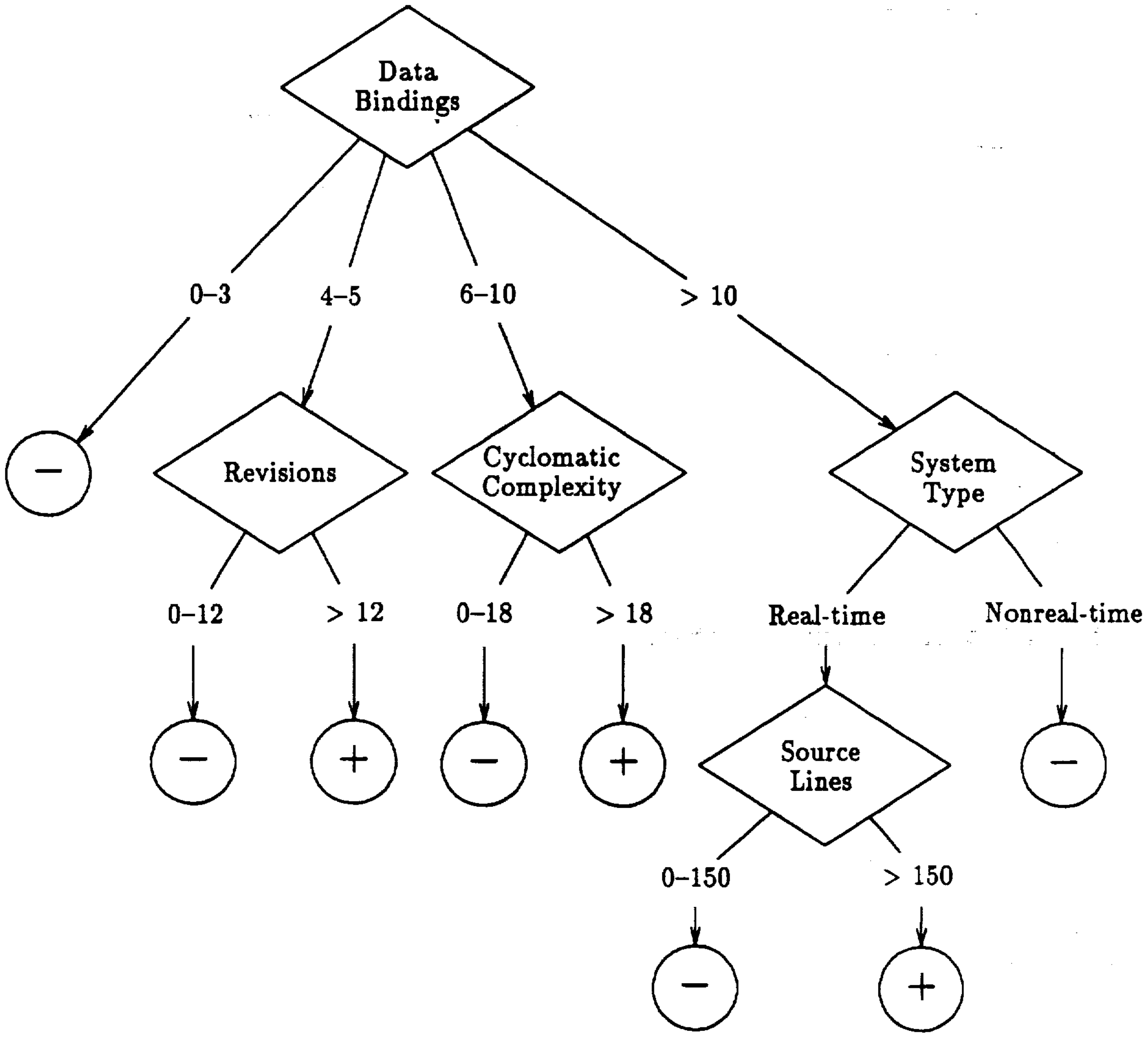
The "80:20 rule" constitutes a fundamental principle in software engineering. The 80:20 rule states that approximately 20 percent of a software system is responsible for 80 percent of its errors and costs. The impact of this rule is especially problematic in large-scale software systems, where it is difficult to determine the complex interrelationships among the large numbers of components. The identification of the high-risk components, i.e. the troublesome 20 percent, provides several benefits to developers:

- localizes components with low reliability;
- focuses testing efforts;
- focuses re-design and re-implementation efforts; and
- facilitates scheduling, among others.

2 Classification Trees

Classification trees provide an approach for identifying high-risk components [Boe81]. The trees use software metrics [Bas80] to classify components according to their likelihood of having certain high-risk properties. Classification trees enable developers to orchestrate the use of several metrics, and hence, they serve as *metric integration frameworks*. Example metrics that may be used in a classification tree are: source lines, data bindings, cyclomatic complexity, data bindings per 100 source lines, and number of data objects referenced. A hypothetical tree appears in Figure 1. Developers can select which high-risk properties interest them. For example, developers may want to identify those components whose: (a) error rates are likely to be above 30 errors per 1000 source lines; (b) error rates are likely to be below 5 errors per 1000 source lines; (c) total error counts are likely to be above 10; (d) maintenance costs are likely to require between 25 to 50 person-hours of effort; (e) maintenance costs are likely to require between 0 to 10 person-hours of effort; or (f) error counts of error type X are likely to be above 0 (e.g., X = interface, initialization, control).

Each of these properties define a "target class," which is the set of components likely to have that property. A classification tree would be generated to classify components as to whether or not they are in each of these target classes. The classification trees are automatically generated using data from previous releases and projects. Classification tree generation is based on a recursive algorithm that selects metrics that best differentiate between components that are and are not



"+" = Classified as likely to have errors of type X

"−" = Classified as unlikely to have errors of type X

Figure 1: Example (hypothetical) software metric classification tree. There is one metric at each diamond-shaped decision node. Each decision outcome corresponds to a range of possible metric values. Leaf nodes indicate whether or not an object is likely to have some property, such as high error-proneness or errors in a certain class.

within a target class [SP88]. A developer wishing to focus resources on high-payoff areas might use several classification trees in support his analysis process. Figure 2 gives an overview of the generation and use of classification trees.

The metric-based classification tree approach has several benefits.

- Users can specify the target classes of components they want to identify.
- Classification trees are *generated automatically* using past data.
- The trees are extensible — new metrics can be added.
- The trees serve as metric integration frameworks — they use multiple metrics simultaneously to identify a particular target class, and may incorporate any metric from all four measurement abstractions: nominal, ordinal, interval, and ratio.
- Classification trees prioritize data collection efforts and quantify diminishing marginal returns.
- The tree generation algorithms are calibratable to new projects and environments using "training sets."
- The tree generation algorithms are applicable to large-scale systems, as opposed to being limited to small-scale applications.

3 Empirically-Based Classification Paradigm

An overview of the classification paradigm appears in Figure 2. The paradigm has been applied in two validation studies using data from NASA [SP88] and Hughes [SP89]. The three central activities in the paradigm are: (i) data management and calibration, (ii) classification tree generation, and (iii) analysis and feedback of newly acquired information to the current project. Note that the process outlined in Figure 2 is an iterative paradigm. The automated nature of the classification tree approach allows classification trees to be easily built and evaluated at many points in the lifecycle of an evolving software project, providing frequent feedback concerning the state of the software product.

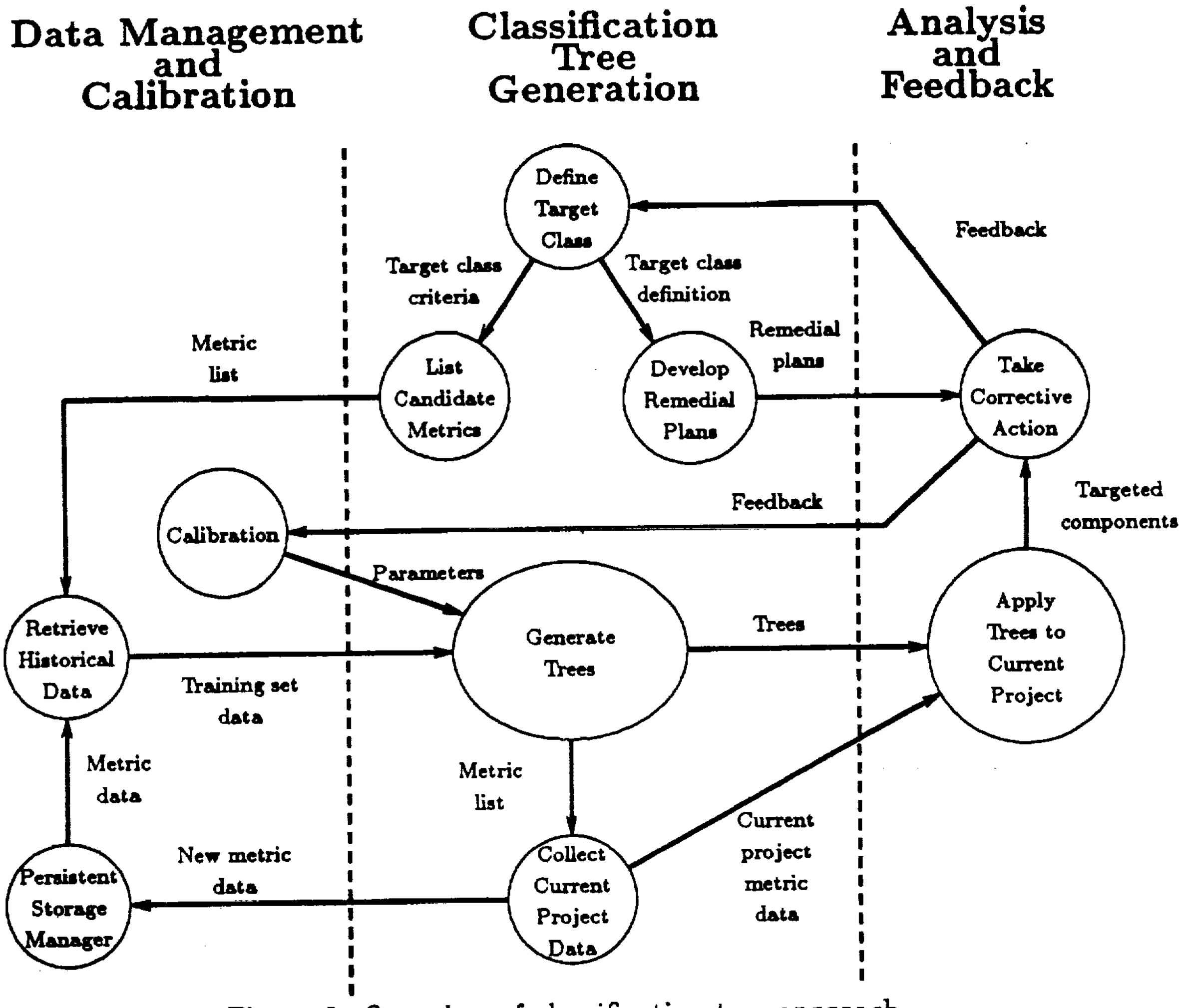


Figure 2: Overview of classification tree approach.

3.1 Classification tree generation

This central activity focuses on the activities necessary to construct classification trees and prepare for later analysis and feedback. During this phase the target classes to be characterized by the trees are defined. Criteria are established to differentiate between members and non-members of the target classes. For example, a target class such as error-prone modules could be defined as those modules whose total errors are in the upper 10 percent relative to historical data. A list of metrics to be used as candidates for inclusion in the classification trees is passed to the historical data retrieval process. A common default metric list is all metrics for which data is available from previous releases and projects.

Importantly, one must determine the remedial actions to apply to those components identified as likely to be members of the target class. For example, if a developer wants to identify components likely to contain a particular type of error, then he should prescribe the application of testing or analysis techniques that are designed to detect errors of that type. Another example of a remedial plan is to consider redesign or reimplementation of the components. It is important to develop these plans early in the process rather than apply ad hoc decisions at a later stage.

Metric data from previous releases and projects as well as various calibration parameters are fed into the classification tree generation algorithms [SP88]. The tree construction process develops characterizations of components within and outside the target class based on measurable attributes of past components in those categories. Classification trees may incorporate metrics capturing component features and interrelationships, as well as those capturing the process and environment in which the components were constructed. Collection of the metrics used in the decision nodes of the classification trees should begin for the components in the current project. This data is stored for future use and passed, along with the classification trees, to the analysis and feedback activity.

3.2 Data management and calibration

Data management and calibration activities concentrate on the retention and manipulation of historical data as well as the tailoring of classification tree parameters to the current development environment. The tree generation parameters, such as the sensitivity of the tree termination criteria, need to be calibrated to a particular environment. For further discussion of generation parameters and examples of how to calibrate them, see [SP88] and [SP89]. Classification trees are built based on metric values for a group of previously developed components, which is called a "training set." Metric values for the training set, as well as those for the current project, are retained in a persistent storage manager.

3.3 Analysis and feedback

In this portion of the paradigm, the information resulting from the classification tree application is leveraged by the development process. The metric data collected for components in the current project is fed into the classification trees to identify components likely to be in the target class. The remedial plans developed earlier should now be applied to those targeted components. When the remedial plans are being applied, insights may result regarding new target classes to identify and further fine tuning of the generation parameters.

4 Validation Studies Using Classification Trees

One validation study has been conducted and another is underway using the classification tree approach. The goal of the studies was to determine the feasibility of the approach and to analyze tree accuracy, complexity, and composition. The first validation study we conducted was using 16 NASA projects (3000–112,000 lines) [BZM+77] [CMP+82] [SEL82]. A total of 9600 classification trees was automatically generated and evaluated based on several parameters. On the average, the trees correctly classified 79.3 percent of the software modules according to whether or not they were in the target classes (see Figure 3) [SP88]. In a second study, we have applied the approach in a Hughes maintenance environment to identify fault-prone and change-prone components in a large-scale system (>100,000 lines) [SP89]. The use of project data from NASA and Hughes is intended to demonstrate the applicability of the method to large-scale projects. The classification tree generation tools are environment independent and are calibrated to particular environments by measurements of past releases and projects.

Basically, a classification tree is a predictive tool that integrates multiple metrics to determine whether or not a module is likely to be in a user-specified "target class." The target classes examined were: cost-prone (NASA only), fault-prone (both environments), and change-prone (Hughes only). Cost-prone was defined as having development costs in the uppermost quartile (i.e., top 25 percent) relative to past data; the definitions were analogous for fault-prone and change-prone. The goal of a classification tree is to identify the modules on a future project that are likely to be in a target class.

For complete descriptions of the studies and examples of the classification trees generated, see [SP88] and [SP89]. The predictive accuracy of the trees in these two environments is summarized in Figures 3 and 4. The purpose of this analysis is not to compare or evaluate the environments in any way — the purpose is to refine and enhance the classification tree technique and underlying concepts. The

Target class	Number of trees	Accuracy: overall (%)	
	N	Mean	Std.
Cost-prone	4800	79.88	14.21
Fault-prone	4800	78.75	21.23
All	9600	79.32	18.07

Figure 3: Classification tree accuracy using metric data from 16 NASA projects.

Target class	Number of trees	Accuracy: overall (%)		Accuracy: consistency (%)		Accuracy: completeness (%)	
	N	Mean	Std.	Mean	Std.	Mean	Std.
Change-prone	10	89.54	5.86	53.12	39.21	90.13	12.90
Fault-prone	10	80.76	14.31	81.16	25.91	75.68	12.61
All	20	85.15	11.56	67.14	35.40	82.91	14.46

Figure 4: Classification tree accuracy using metric data from the Hughes project.

classification trees in both environments are relatively accurate overall. Several measures of accuracy are being examined, including

1. **Overall accuracy.** Percentage of future project modules correctly predicted by the classification tree (i.e., target class modules correctly predicted as such and non-target class modules correctly predicted as such). (Calculated in both NASA and Hughes studies.)

2. **Completeness.** Percentage of target class modules in the future project that were predicted as such by the classification tree. (Calculated in Hughes study only.)

3. **Consistency.** Percentage of future project modules predicted as target class modules by the classification tree that actually were target class modules. (Calculated in Hughes study only.)

The overall accuracy of the classification tree depends on the target class being identified (see Figures 3 and 4). The classification trees are more accurate for identifying change-prone files than fault-prone files in the Hughes study (Figure 4). When accuracy is measured in terms of consistency and completeness, the classification trees have 83 percent completeness and 67 percent consistency (Figure 4). Therefore, the trees tend to identify correctly most (83 percent) of the targeted files, but they also raise several false alarms (33 percent of the files predicted to be in the target class are not in the target class). Different environments may assign different priorities to consistency and completeness accuracy measures. Note that there is not necessarily a tradeoff between the two — it is possible for classification trees to have both high consistency and high completeness. The trees targeting change-prone files were 90 percent complete, and those targeting fault-prone files were 81 percent consistent (Figure 4). Based on the Hughes data (Figure 4), when developers invest additional effort in the modules identified as likely to be fault-prone, they have relatively high assurance that their resources will be well spent (since those trees have high consistency). When developers invest additional effort in the files identified as likely to be change-prone, they have relatively high assurance that most of the change-prone files have been identified (since those trees have high completeness).

5 Classification Tree Tools and the *Amadeus* System

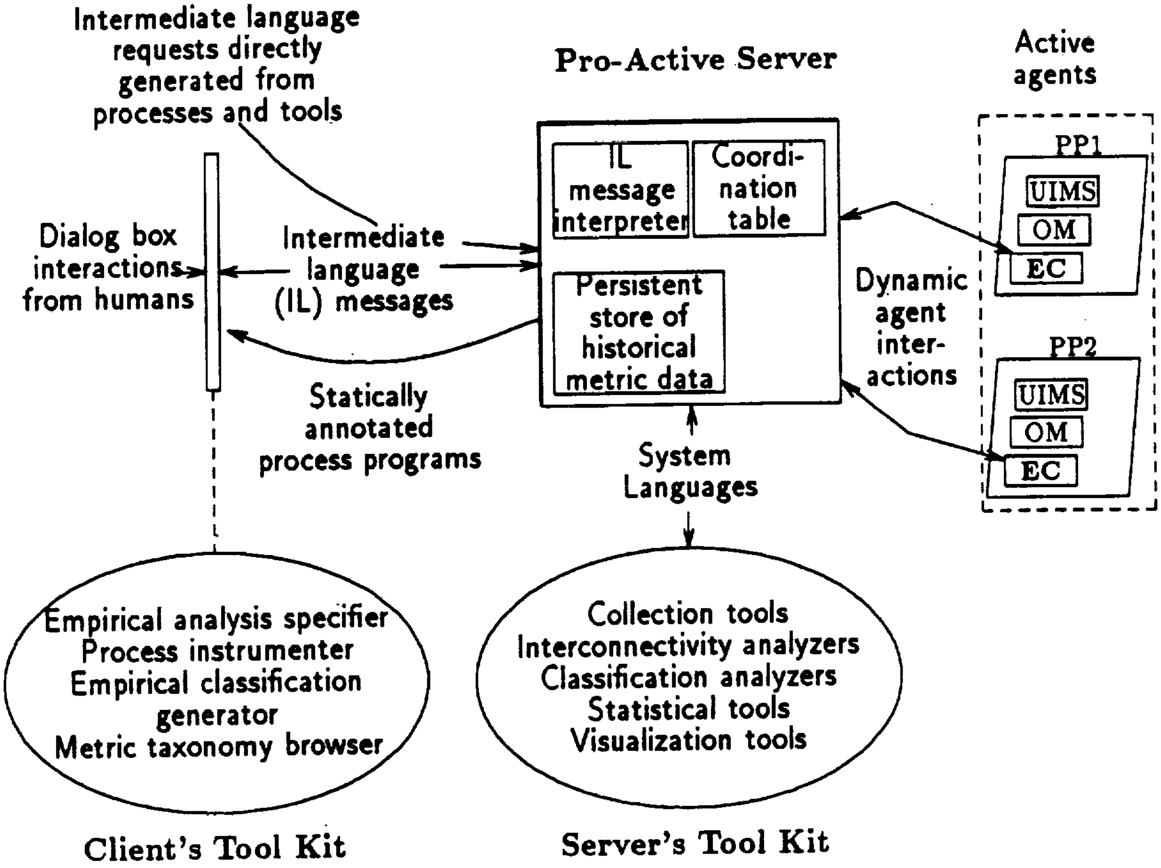
Preliminary tool prototypes have been developed to automatically generate metric-based classification trees. These tools embody the classification tree generation algorithms and supporting data manipulation capabilities. These tools are prototypes and should be considered only preliminary versions.

The classification tree tools will become part of the *Amadeus* system, which is an automated measurement and empirical analysis system under development at the University of California at Irvine. *Amadeus* supports empirically-based analysis techniques for use in the development and evolution of large-scale software. Empirically-based techniques use measurements to describe, analyze, and control software systems and their development processes. These modeling techniques leverage past experience and have many desirable properties, including being scalable to large systems, integratable, and calibratable to new projects. The *Amadeus* system provides capabilities for specifying empirical analyses, collecting the underlying data, and feeding the results back into the development processes. *Amadeus* serves as an extensible integration framework for empirically-based analysis techniques, and hence, it is a complementary project to the *TAME* project at the University of Maryland [BR88]. *Amadeus* is integrated with and leverages the components in the *Arcadia* software environment architecture [TBC+88]. The conceptual architecture of the *Amadeus* system appears in Figure 5.

6 Conclusions

This paper has presented an analysis of a software metric classification tree approach to the problem of localizing fault-prone, cost-prone, and change-prone software components. Classification trees have the benefit of being general structures that can be automatically generated and evaluated and naturally decomposed into a set of if-then rules. The metric evaluation heuristics can result in relatively rapid construction of the trees, as they did in this analysis, but we do not imply that this is the case for all heuristics and data sets. The empirical results presented in this study are intended to provide the basis for analysis of classification tree generation and evaluation — it is not implied that there is a direct extrapolation of these results to other environments and data sets.

Classification trees enable developers to leverage off the use of multiple metrics. In related metrics work (e.g., [BR88]), a variety of individual metric collection tools are being developed, such as tools for data bindings metrics, cyclomatic complex-


Intermediate language requests directly generated from processes and tools
Pro-Active Server
Active agents
IL message interpreter
Coordination table
Persistent store of historical metric data
PP1
UIMS
OM
EC
PP2
UIMS
OM
EC
Dialog box interactions from humans
Intermediate language (IL) messages
Statically annotated process programs
Dynamic agent interactions
System Languages
Empirical analysis specifier
Process instrumenter
Empirical classification generator
Metric taxonomy browser
Collection tools
Interconnectivity analyzers
Classification analyzers
Statistical tools
Visualization tools
Client's Tool Kit
Server's Tool Kit

IL messages = Condition-action pairs. Conditions: event-based, object-based, or time-based. Actions: processes or tools.

Server = Pro-active server interprets IL requests, delegates dynamic collection to individual EC's, and coordinates analysis across multiple EC's. Server is PPL, UIMS, and OM independent.

EC = Evaluation component in active agent. EC is PPL, UIMS, and OM dependent.

Figure 5: Conceptual architecture of the *Amadeus* measurement and empirical analysis system.

ity, source lines, change history, and error history. Classification trees provide a mechanism for integrating the metric data resulting from these tools. An extensive validation study has been conducted using classification trees and another is underway.

Further research is underway to expand the scope of analysis and to refine the underlying principles driving the results. The accuracy measures of completeness and consistency are being applied to the NASA project data. The components that are chronically misclassified are being characterized in order to guide the definition of new metrics. The use of project data from NASA and Hughes is intended to demonstrate the applicability of the method to large-scale projects. The tree generation tools are environment independent and are calibrated to particular environments by measurements of past releases and projects. Classification analysis, along with other empirically-based analysis techniques for large-scale software, will be supported in the *Amadeus* measurement and empirical analysis system.

References

[Bas80] V. R. Basili. *Tutorial on Models and Metrics for Software Management and Engineering.* IEEE Computer Society, New York, 1980. IEEE Catalog No. EHO-167-7.

[Boe81] B. W. Boehm. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[BR88] V. R. Basili and H. D. Rombach. The tame project: Towards improvement-oriented software environments. *IEEE Trans. Software Engr.*, SE-14(6):758–773, June 1988.

[BZM+77] V. R. Basili, M. V. Zelkowitz, F. E. McGarry, Jr. R. W. Reiter, W. F. Truszkowski, and D. L. Weiss. The software engineering laboratory. Technical Report SEL-77-001, Software Engineering Laboratory, NASA/Goddard Space Flight Center, Greenbelt, MD, May 1977.

[CMP+82] D. N. Card, F. E. McGarry, J. Page, S. Eslinger, and V. R. Basili. The software engineering laboratory. Technical Report SEL-81-104, Software Engineering Laboratory, NASA/Goddard Space Flight Center, Greenbelt, MD, Feb. 1982.

[SEL82] SEL. Annotated bibliography of software engineering laboratory (sel)literature. Technical Report SEL-82-006, Software Engineering Laboratory, NASA/Goddard Space Flight Center, Greenbelt, MD, Nov. 1982.

[SP88] R. W. Selby and A. A. Porter. Learning from examples: Generation and evaluation of decision trees for software resource analysis. *IEEE Trans. Software Engr.*, 14(12):1743–1757, December 1988.

[SP89] R. W. Selby and A. A. Porter. Software metric classification trees help guide the maintenance of large-scale systems. In *Proceedings of the Conference on Software Maintenance*, pages 116–123, Miami, FL, October 1989.

[TBC+88] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michal Young. Foundations for the Arcadia environment architecture. In *Proceedings of ACM SIGSOFT '88: Third Symposium on Software Development Environments*, pages 1–13, Boston, November 1988. Appeared as *Sigplan Notices 24*(2) and *Software Engineering Notes 13*(5).

VIEWGRAPH MATERIALS

FOR THE

R. W. SELBY PRESENTATION

Classification Tree Analysis Using the *Amadeus* Measurement and Empirical Analysis System

Richard Selby, Greg James
Kent Madsen, Joan Mahoney
Adam Porter, and Douglas Schmidt

Department of Computer Science
University of California
Irvine, California 92717

Research Vision

- Long-term goal: Develop, evaluate, and integrate empirically-based analysis techniques for catalyzing the development and evolution of large-scale software
  —> *Empirically-guided software development*

- Research areas:
  - Empirically-guided process programs
  - Classification trees as integration mechanisms for metrics
  - Interconnectivity analysis methods
  - Environment components to support measurement and empirical analysis

Why Empirically-Based Techniques?

- Scalable to large projects

- Calibratable to new environments

- Measurements are integratable

- Leverage previous experience

The *Amadeus* System

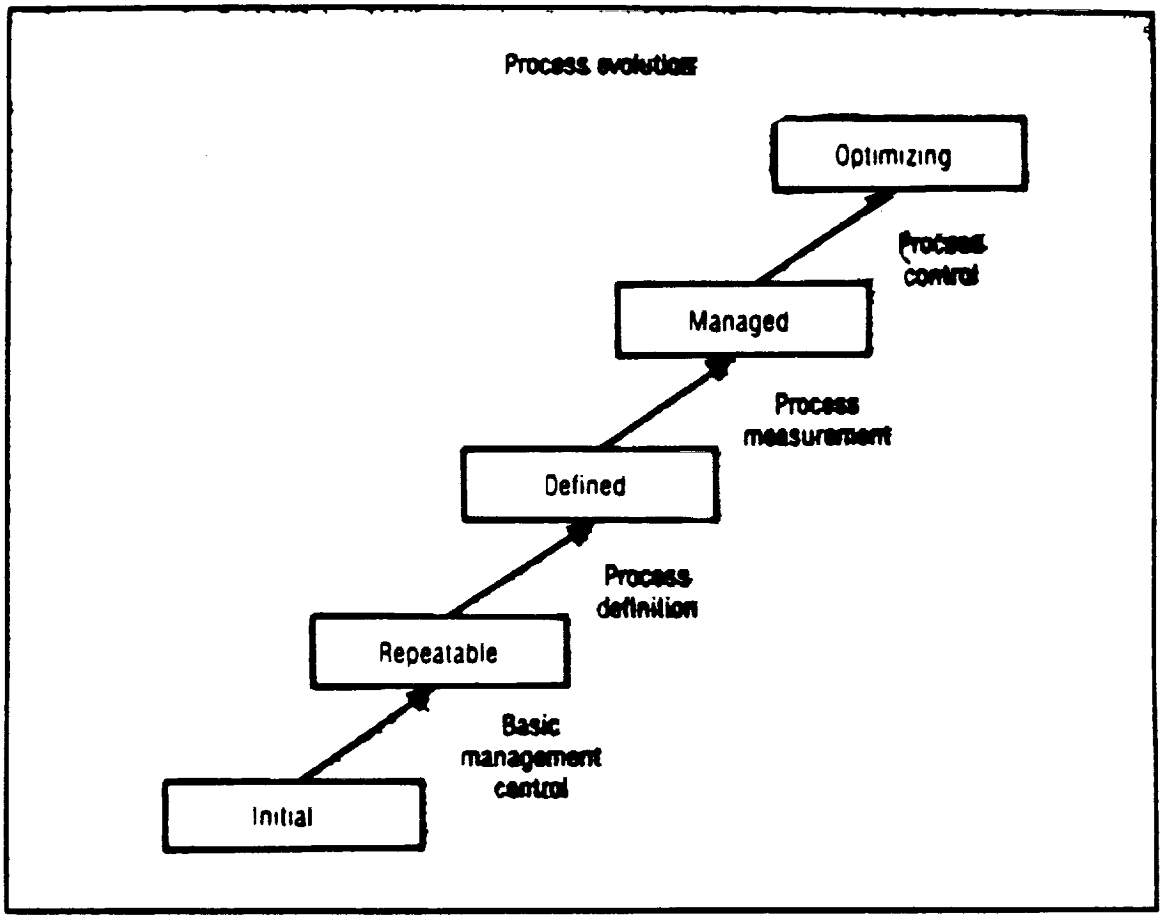
- The *Arcadia* measurement and empirical analysis system

- Integrated with and leverages the components of the *Arcadia* software environment architecture

- *Amadeus* provides an extensible integration framework for empirically-based analysis techniques

- *Amadeus* focuses on support for large-scale software objects and processes

Amadeus Supports Process Maturity Levels 4 and 5: Managed and Optimizing Processes (Humphrey)



The five levels of process maturity.

ORIGINAL PAGE IS
OF POOR QUALITY

Amadeus Major Functional Areas and Example Capabilities

- Definition
  - Empirical analysis specifier, metric taxonomy browser
- Collection
  - Event-based, object-based, time-based mechanisms
- Analysis
  - Classification tree generator, experimental design builder, data analysis tools
- Feedback
  - Empirical specification generator

Amadeus System

- Client/server separation

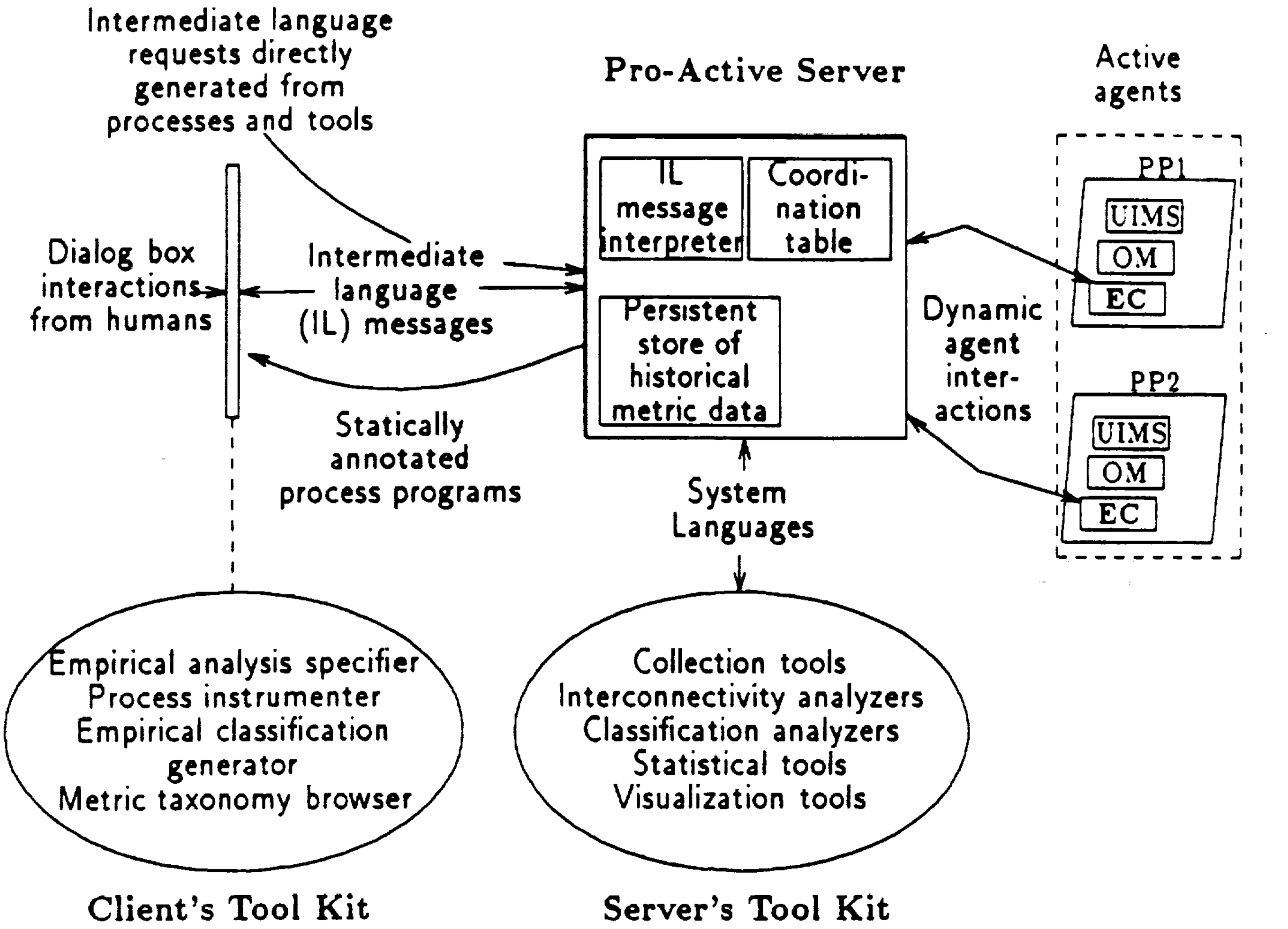
- Client "toolkit"

- Server "toolkit"

- Intermediate language (IL)

- Evaluation component (EC)

Amadeus Environment Components: Conceptual Model


Intermediate language requests directly generated from processes and tools
Pro-Active Server
Active agents
IL message interpreter
Coordi- nation table
PP1
UIMS
OM
EC
Dialog box interactions from humans
Intermediate language (IL) messages
Persistent store of historical metric data
Dynamic agent inter- actions
PP2
UIMS
OM
EC
Statically annotated process programs
System Languages
Empirical analysis specifier
Process instrumenter
Empirical classification generator
Metric taxonomy browser
Collection tools
Interconnectivity analyzers
Classification analyzers
Statistical tools
Visualization tools
Client's Tool Kit
Server's Tool Kit

IL messages = Condition-action pairs. Conditions: event-based, object-based, or time-based. Actions: processes or tools.

Server = Pro-active server interprets IL requests, delegates dynamic collection to individual EC's, and coordinates analysis across multiple EC's. Server is PPL, UIMS, and OM independent.

EC = Evaluation component in active agent. EC is PPL, UIMS, and OM dependent.

Metric-Based Classification Trees

- Motivated by the "80:20 rule"

- 80:20 rule is especially problematic in large-scale systems

- Goal is to identify the high-risk components, i.e., the "troublesome 20 percent"

- Classification trees use *multiple metrics* to identify these components

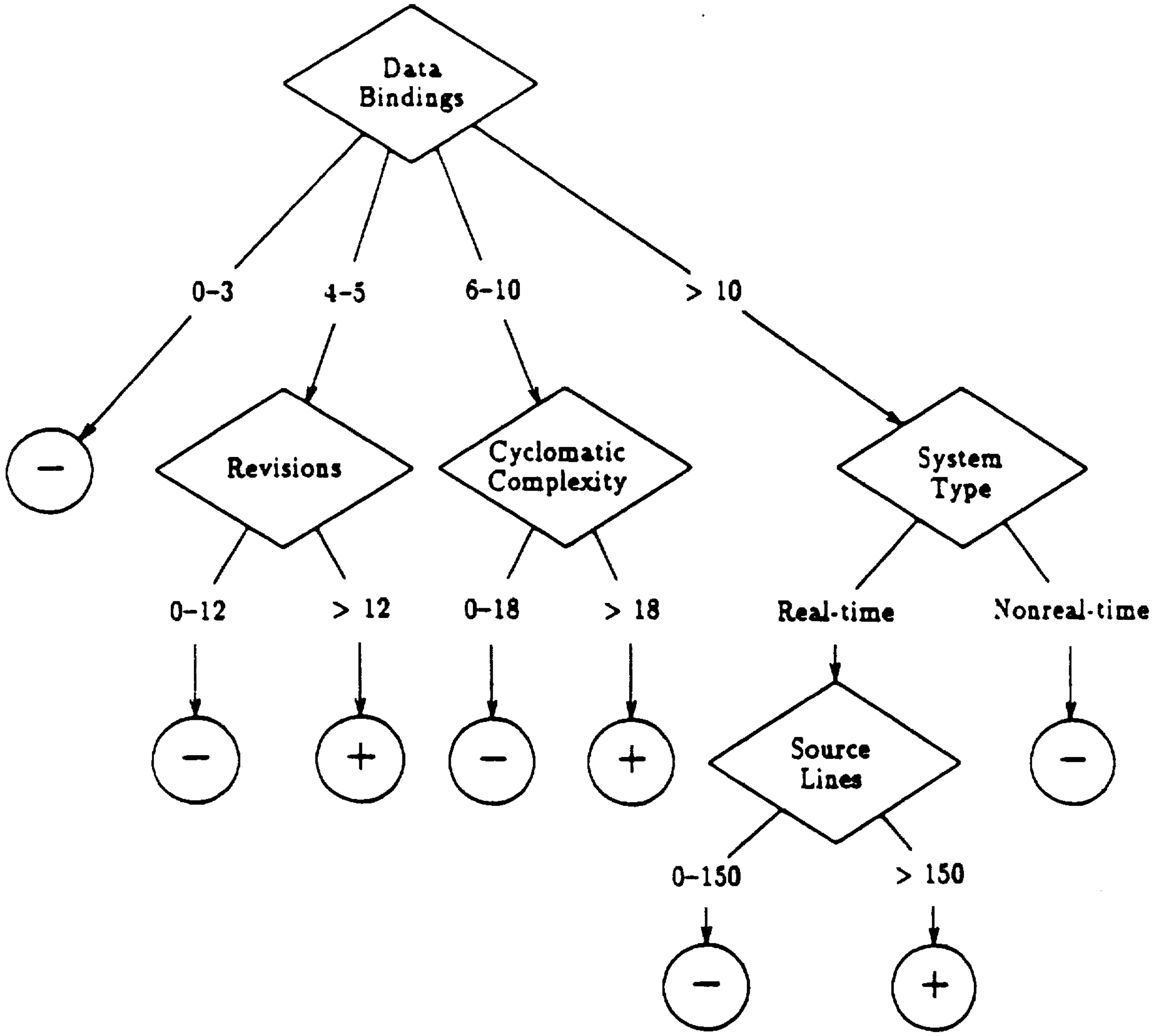
- Classification trees use metrics to classify components according to their likelihood of having certain high-risk properties

Example "Target Classes" to Identify

Components whose:

- *Error rates* are likely to be above 30 errors/KLOC
- *Error rates* are likely to be below 5 errors/KLOC
- *Total error counts* are likely to be above 10
- *Maintenance costs* are likely to require between 25 to 50 person-hours of effort
- *Maintenance costs* are likely to require between 0 to 10 person-hours of effort
- *Error counts of error type X* are likely to be above 0

Example Hypothetical Classification Tree



"+" = Classified as likely to have errors of type *X*

"−" = Classified as unlikely to have errors of type *X*

R.W. Selby
U.C. Irvine

Benefits of Classification Trees

- *User-specifiable* target classes of components to identify

- Trees *generated automatically* using past data

- *Extensible* trees — new metrics can be added

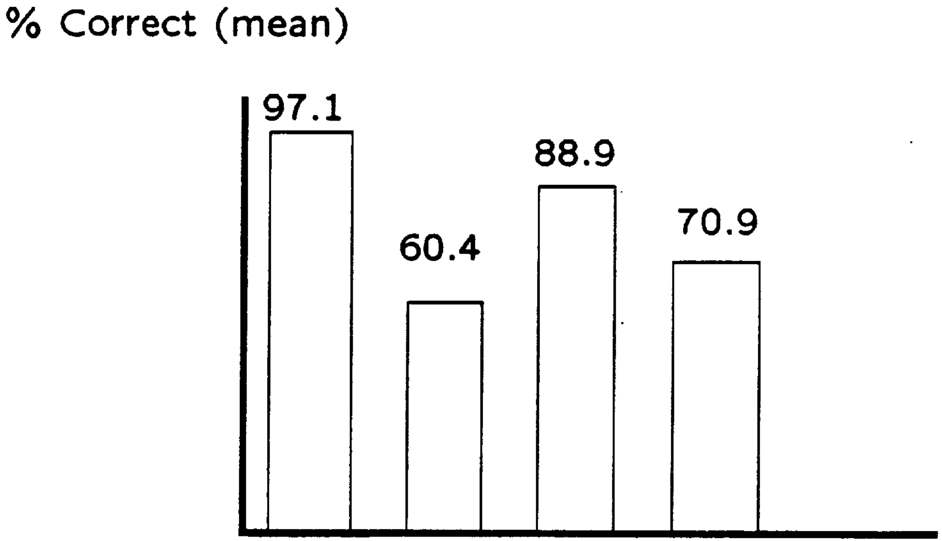
- Trees serve as *integration frameworks* for multiple metrics; may incorporate any metric from all four measurement abstractions

- Trees *prioritize data collection* efforts

- Tree generation algorithms are *calibratable* to new projects and environments

- Algorithms applicable to *large-scale systems*

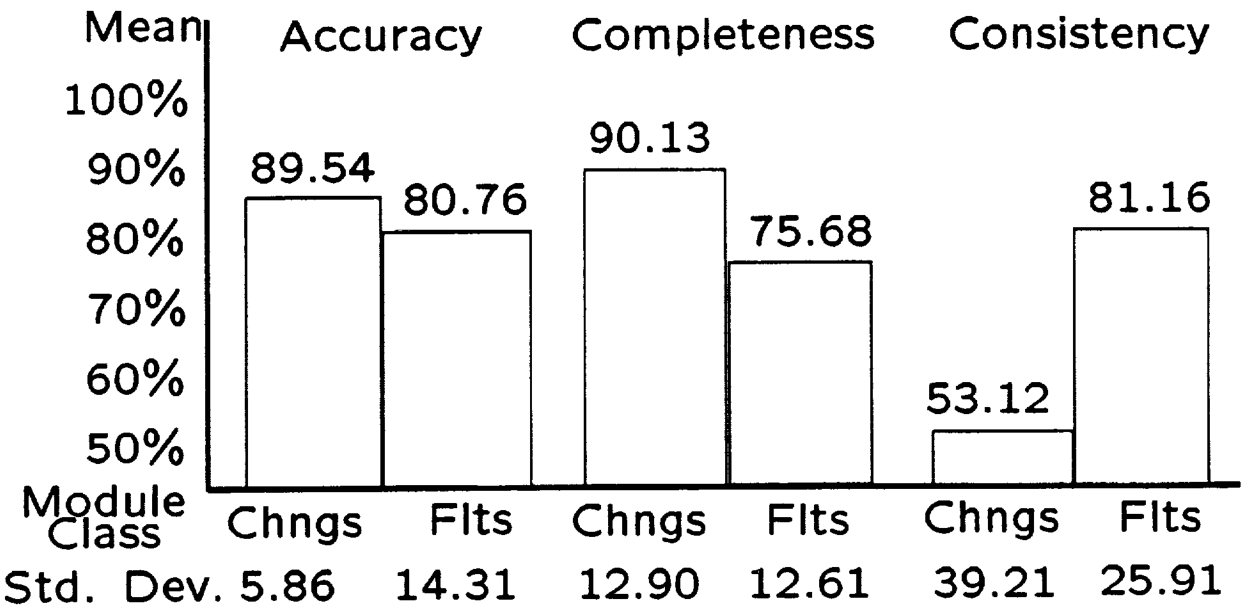
Classification Tree Accuracy



Attribute Availability	all	early	all	early
Module class	faults	faults	total effort	total effort
Std. Dev.	6.72	13.50	6.17	14.23

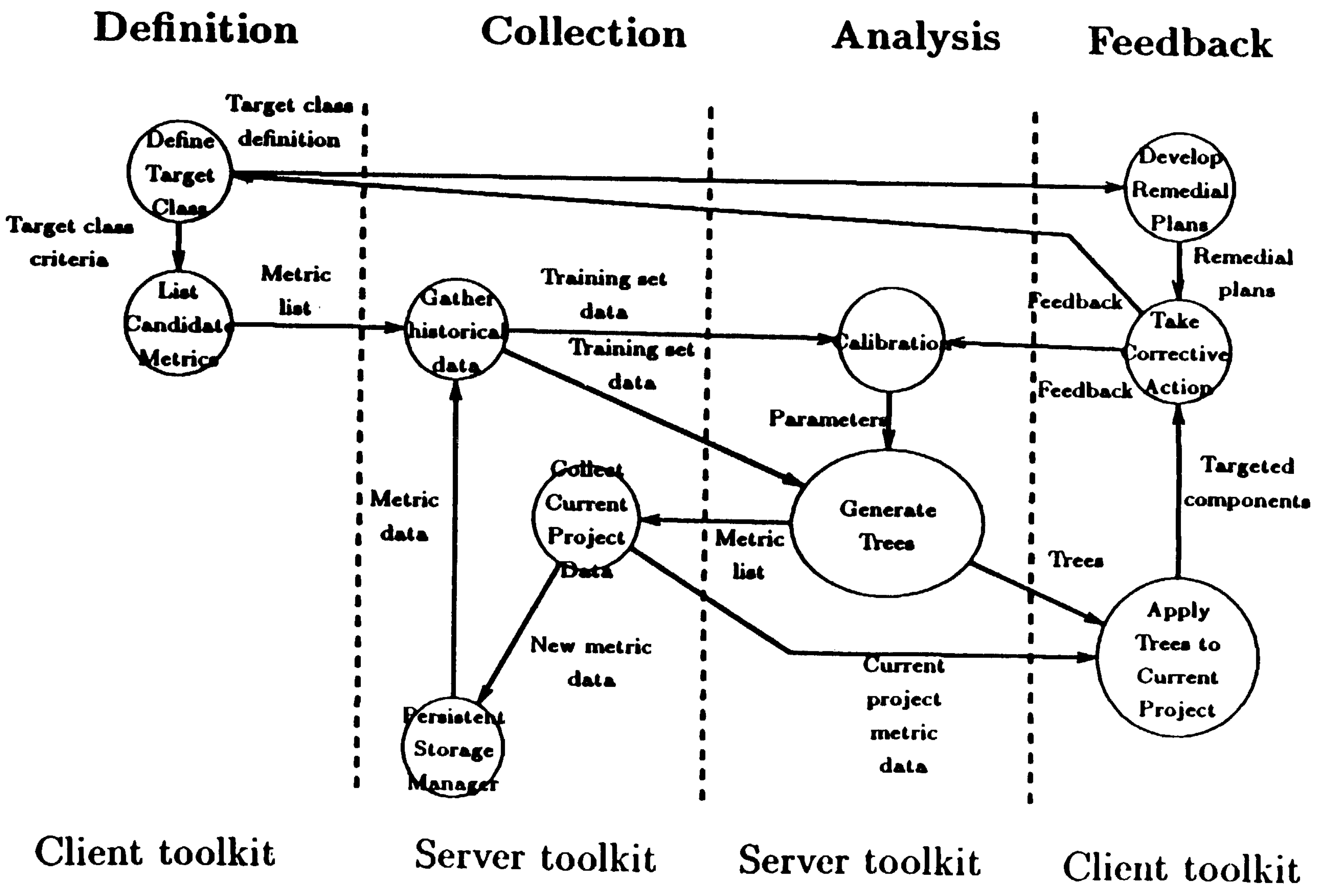
- Average of 79.3% (sd = 18.1) modules correctly identified over all 9600 tree combinations
- Better identification of costly modules (79.9%) vs. error-prone modules (78.8%) [p <.0003]*
- Better identification using all metrics (93.0%) vs. early metrics (65.6%)
- Early metrics better for identification of costly modules

Preliminary Results



- The classification trees are more accurate for identifying change-prone files than fault-prone files.
- The classification trees have 83 percent completeness and 67 percent consistency
- The trees targeting change-prone files were 90 percent complete.
- The trees targeting fault-prone files were 81 percent consistent.

Classification Methodology



Conclusions

- Identifying high-risk components (e.g., fault-prone) benefits development personnel

- Automatic generation of metric-based classification trees has shown merit in two "proof of concept" studies

- Further research on generation algorithms underway

- Classification analysis will be supported in the *Amadeus* measurement and empirical analysis system

ABSTRACT

The Jet Propulsion Laboratory's Experience with Formal Inspections

Marilyn Bush and John Kelly
Jet Propulsion Laboratory

INTRODUCTION

This paper describes the introduction of software formal inspections, known in the literature as Fagan Inspections, to the Jet Propulsion Laboratory by the JPL Software Product Assurance Section. It briefly describes the nature of the inspections, indicates the way they altered JPL practice, characterizes their present status at JPL, and evaluates their initial impact.

WHY SOFTWARE PRODUCT ASSURANCE INTRODUCED FORMAL INSPECTIONS

The Jet Propulsion Laboratory(JPL), a division of the California Institute of Technology, is responsible to NASA for conducting scientific investigations of the solar system using automated spacecraft. Since the early 1980's JPL has been concerned about the way it manages and develops software systems. At that time, several JPL software systems were experiencing difficulty, even as it was becoming evident that such systems would become a larger part of NASA's future. By the mid 1980's software has the focus of about 50% of JPL's work hours. By the year 2000, some estimates show software efforts rising to as much as 80%. Software problems are especially important to NASA because critical flight software must be error free.

SPA soon learned that spending a little money to find and fix defects early in the development life cycle saves a lot of money later. One study, for example, estimates that $100 spent to find and fix a defect during the requirements phase saves $10,000 to find and fix the same defect in the operations phase.

In surveying what worked best in the most efficient software operations around the country, it was determined that the most cost-effective early defect detection technique was "Fagan Inspections."

Formal Inspections were originated at IBM by Michael Fagan, 1976. Formal Inspections represent a defect detection, removal, and correction verification process carried out during the pretest phases of the development lifecycle. Formal Inspections were found to be very effective at ensuring that documents and code are logically correct, complete, clear, reliable and consistent.

Since 1976, Formal Inspections have spread widely throughout IBM, then to other leading software producing organizations. The Software Engineering Institute recommends the use of formal inspections for enhancing developers' capability to consistently produce high quality software. JPL's Software Product Assurance Section has been aggressive in implementing the inspection process on several projects.

WHAT FORMAL INSPECTIONS ARE

Formal Inspections are a seven-step process to find, fix, and document defects early in the development cycle. The steps are planning, overview, preparation, inspection meeting, third-hour, rework, and follow-up. The inspection team consists of a trained moderator and peers (3-6 people), with defined roles, representing areas of the project affected by material being inspected. No managers are present in the actual inspection meeting that last no more than two hours. Inspections are carried out within designated phases of the software life cycle. At JPL, the phases include System Requirements, Subsystem Requirements, Functional Design, Software Requirements, Architectural Design, Detailed Design, Source Code, Test Plan, and Test Procedures.

Checklists of tailored questions are used to identify defects. Statistics on the number of defects, types of defects, and time expended on inspections are kept. Defects are categorized as MAJOR (which must be fixed immediately) and MINOR (which are fixed at the discretion of the project). Major defects, if they were allowed to remain, would result in the system not operating as required.

HOW FORMAL INSPECTIONS WERE INTRODUCED AT JPL

SPA introduced the idea of Formal Inspections to JPL by both training managers in the value of inspections, and at the same time training developers in using inspections. We spent six months developing two training courses: one for developers and one for managers. The 12-hour developer course (which includ ; moderator training) was completed in February 1988. The two-hour manager course was developed in June 1988. Moderators not only take the 12-hour developer course, but are observed during their first two inspections. There are also monthly moderator meetings. As of September 1989, thirteen "manager" and 20 "developer" classes have been held, involving 175 managers and 288 technical staff, making a total of 463 trained JPL people.

We found after conducting 171 inspections that JPL's averages per inspection were as follows:

Major Defects found	3.6
Minor Defects found	12.2
Total Staff hours	27.6
Pages inspected (over the 2 hr meeting)	37.7

The number of participants involved in an inspection did not appear to significantly increase the number of defects found (the teams ranged in size from 3 to 6). This means that inspection teams of 3 may be viable for code inspections. A significant preliminary discovery was that code audits were not nearly as effective at finding defects as code inspections. Even though code inspections uncovered fewer defects than design inspections, the audits found even fewer defects (one-third fewer).

All of the types of inspections tried indicate that Formal Inspections are a cost effective means of improving quality early in a project's life cycle. The average number of work hours needed to find, fix and verify the correction of a defect (major and minor combined) ranged from 1.4 to 1.8 hours.

VIEWGRAPH MATERIALS

FOR THE

M. BUSH PRESENTATION

THE
JET PROPULSION LABORATORY'S EXPERIENCE WITH FORMAL INSPECTIONS

JPL

MARILYN BUSH AND JOHN KELLY

PRESENTATION TO
THE 14TH ANNUAL SOFTWARE ENGINEERING WORKSHOP

NOVEMBER 29, 1989

JPL

CONTEXT

GOAL: TO FIND AND FIX DEFECTS EARLY IN THE DEVELOPMENT LIFE CYCLE.

- $100 SPENT TO FIND AND FIX A DEFECT DURING THE REQUIREMENTS PHASE *CORRESPONDS TO* $10,000 TO FIND AND FIX THE SAME DEFECT DURING THE OPERATIONS PHASE.

*Based on Boehm 1981

JPL

WHAT ARE FORMAL INSPECTIONS?

- Inspections are carried out at designated phases of the software life cycle. Inspections are not substitutes for major milestone reviews.
- Inspections are carried out by peers representing the areas of the life cycle affected by the material being inspected (usually limited to 6 or less people). Everyone participating should have a vested interest in the work product.
- Management is not present during inspections. Inspections are not to be used as a tool to evaluate workers.
- Inspections are led by a trained moderator.

JPL

WHAT ARE FORMAL INSPECTIONS? (CONT.)

- Trained inspectors are assigned specific roles.
- Inspections are carried out in a prescribed series of steps (as described in the *Quick Reference Guide)*.
- Inspection meetings are limited to two hours.
- Checklists of questions are used to define the task and to stimulate defect finding.
- Material is covered during the inspection meeting at a rate which has been found to give maximum error-finding ability.
- Statistics on the number of defects, the types of defects, and the time expended by engineers on inspections, are kept as a historical data base to serve for later trend analysis.

DIFFERENCES BETWEEN FORMAL INSPECTIONS AND WALK-THROUGHS

PROPERTIES	INSPECTION	WALK-THROUGH
•Formal moderator training	Mandatory	Optional
•Definite participant roles	Mandatory	Optional
•Who "drives" the inspection	Moderator	Author
•Use "How to find errors" checklists	Mandatory	Optional
•Use distribution of defects	Mandatory	Optional
•Follow-up to reduce bad fixes	Mandatory	Optional
•Metrics collected to improve effectiveness	Mandatory	Optional

TYPES OF FORMAL INSPECTIONS

- SY SYSTEM REQUIREMENTS
- SU SUBSYSTEM REQUIREMENTS
- R0 FUNCTIONAL DESIGN INSPECTION
- R1 SOFTWARE REQUIREMENTS INSPECTION
- I0 ARCHITECTURAL DESIGN INSPECTION
- I1 DETAILED DESIGN INSPECTION
- I 2 SOURCE CODE INSPECTION
- IT1 TEST PLAN INSPECTION
- IT2 TEST PROCEDURES & FUNCTIONS INSPECTION
- OTHER: OPERATOR'S MANUAL, NEW STANDARDS, PLANS, ETC.

JPL

HOW FORMAL INSPECTIONS WERE INTRODUCED AT JPL

- SPA Informed Managers about the value of Inspections and trained Developers to use Inspections.

  -Over 600 people trained since March 1988.

- "Improving Software Quality" Conference, July 1988 in which Industry Representatives recounted their experience with Inspections.

  -One speaker was Michael Fagan, the inventor of Formal Inspections

- SPA met with every Project Manager and Line Manager to explain the value of Formal Inspections.

  -200 meetings.

- Every JPL-SPA person trained to be a Moderator.

- SPA set up a consultant center for use by projects.

JPL

FINDING DEFECTS IN EARLY PHASES OF THE SOFTWARE DEVELOPMENT LIFE CYCLE

FORMAL INSPECTIONS:

- ARE THE SINGLE MOST IMPORTANT QUALITY IMPROVEMENT TECHNIQUE A DEVELOPER CAN IMPLEMENT, ACCORDING TO EXPERTS AT IBM, BELL LABS, THE SOFTWARE ENGINEERING INSTITUTE AND OTHERS.
- CAN FIND 75-90% OF ALL DEFECTS AT EARLY PHASES OF SOFTWARE DEVELOPMENT.*
- CAN PRODUCE OVERALL PRODUCTIVITY INCREASE OF APPROXIMATELY 14-25%.**
- CAN PRODUCE OVERALL REDUCTION IN CORRECTIVE MAINTENANCE OF 90%.**

*Technical Report, IBM Federal Systems Division, 1986
**Ackerman 1989

JPL STATUS

- SINCE MARCH 1988, CLOSE TO 300 INSPECTIONS HAVE BEEN HELD.
- 10 PROJECTS HAVE ADOPTED FORMAL INSPECTIONS.
- 5 MORE PROJECTS HAVE MADE PLANS TO USE INSPECTIONS.

THE BOTTOM LINE

(AVERAGE PER INSPECTION)

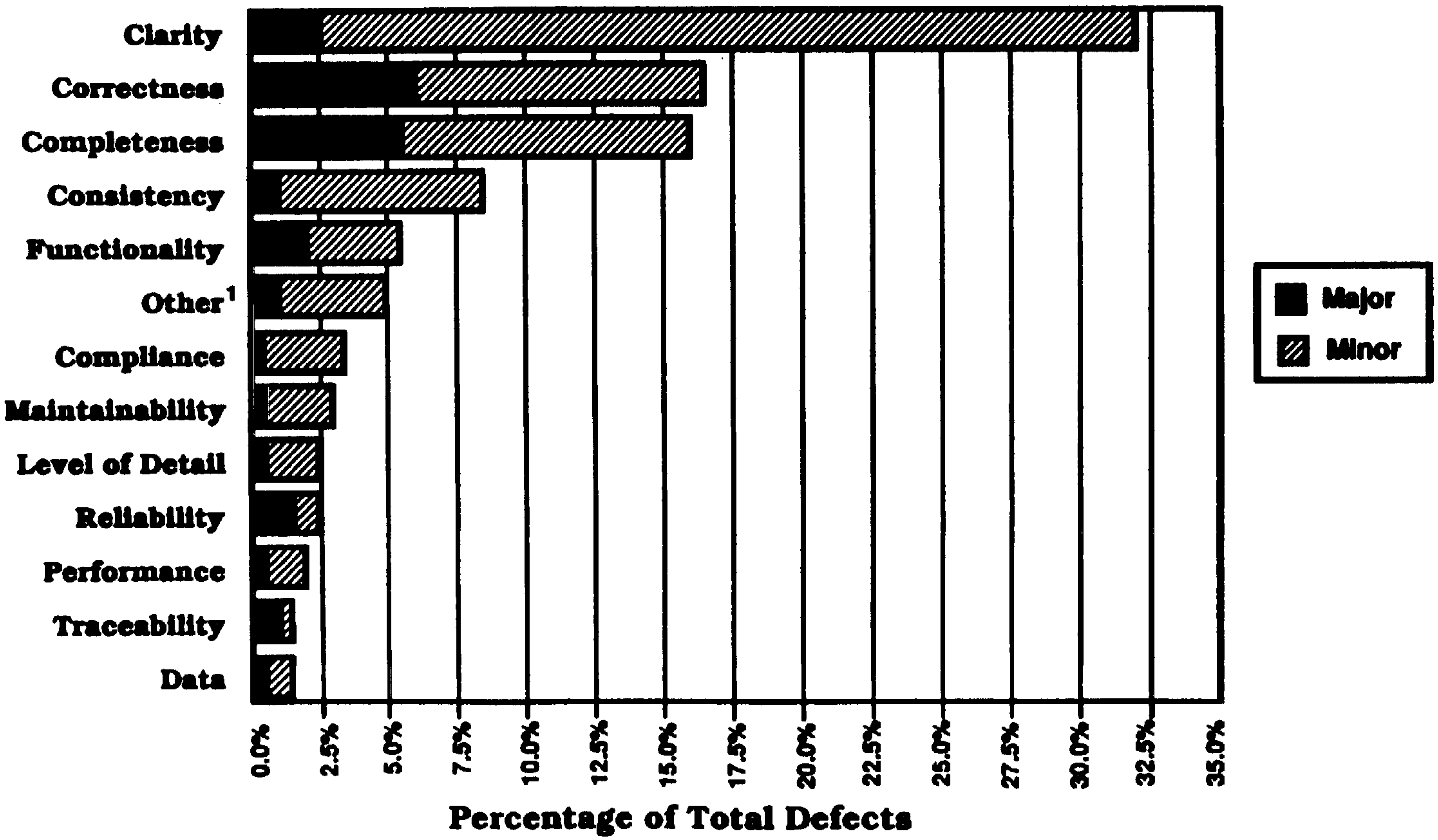
MAJOR DEFECTS FOUND:	4
MINOR DEFECTS FOUND:	12
PAGES COVERED:	38
PARTICIPANTS:	5
TOTAL SPA AND DEVELOPER TIME EXPENDED:	28 HOURS
APPROXIMATE MONEY SAVED BY CORRECTING DEFECTS EARLY:	$25,000

In all JPL has saved $7.5 million over the 300 inspections.

RESULTS
PERCENTAGE OF DEFECT TYPES FOUND

Distribution of Defects by Classification

n = 171 inspections



Software Product Assurance

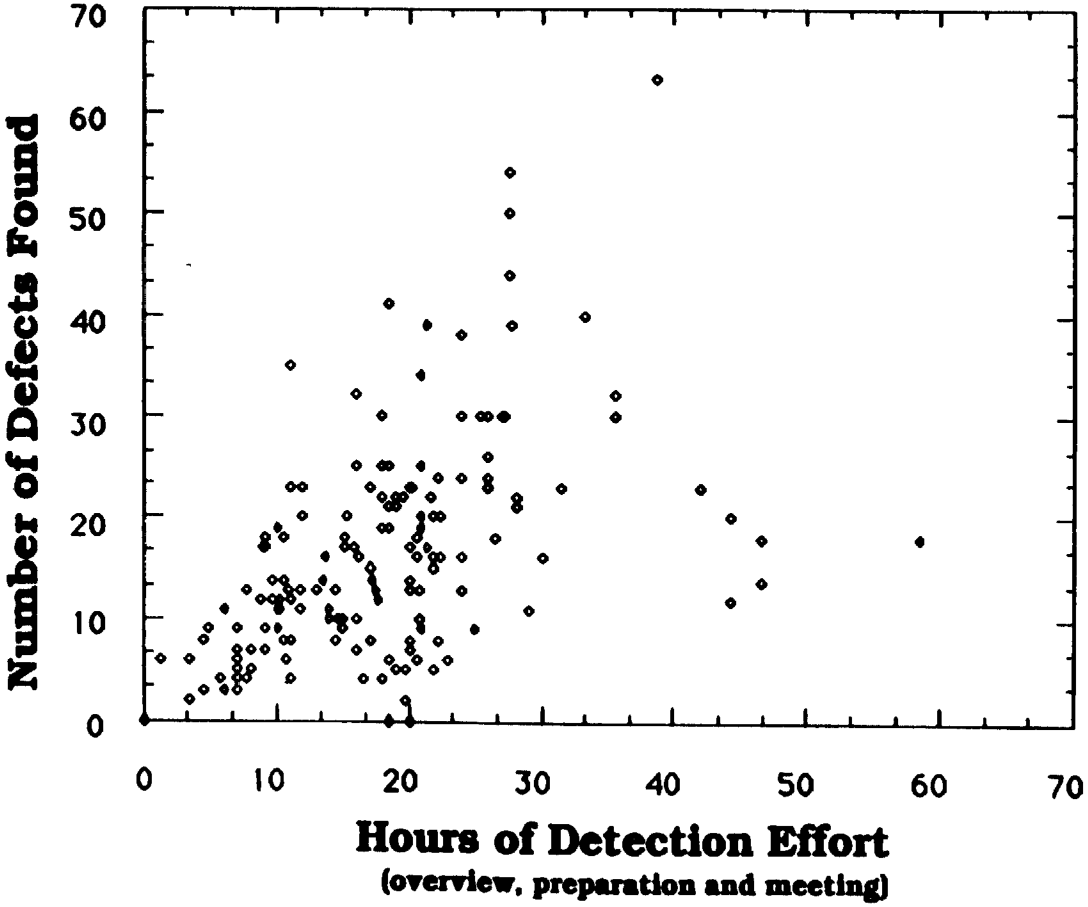
[1] Other includes the classifications with less than 20 defects.

MB8930-9

JPL

RESULTS

Effort vs Number of Defects Found (per inspection)



1: n = 171 inspections

RESULTS

- INSPECTIONS ARE VERY GOOD AT FINDING AND REMOVING THE FOLLOWING CATEGORIES OF DEFECTS: CORRECTNESS, COMPLETENESS, CLARITY, FUNCTIONALITY, RELIABILITY AND CONSISTENCY.

- THE AVERAGE NUMBER OF WORK HOURS TO FIND, FIX, AND VERIFY THE CORRECTION OF A DEFECT VARIES BETWEEN 1.4 AND 1.8 HOURS (AT A COST OF $84 TO $108).

- THE NUMBER OF PARTICIPANTS DOES NOT APPEAR TO HAVE A SIGNIFICANT EFFECT ON THE NUMBER OF DEFECTS FOUND.

- ALL TYPES OF INSPECTIONS ARE COST EFFECTIVE.

LESSONS LEARNED

- TRAINING IS ESSENTIAL FOR SUCCESSFUL FORMAL INSPECTIONS.
- PROJECTS NEED TO PROVIDE ADEQUATE TIME FOR INSPECTIONS.
- MANAGERS NEED TO BE INFORMED ABOUT THE VALUE OF INSPECTIONS.
- HAVE AT LEAST ONE FULL-TIME PERSON ON EACH PROJECT WHOSE PRIMARY RESPONSIBILITY IS THE INSPECTION PROCESS.
- INSPECT CODE BEFORE UNIT TESTING.
- RESCHEDULE INSPECTION MEETING IF PREPARATION TIME IS LESS THAN 5 HOURS.
- NEED TO IMPROVE FIXING MAJOR ERRORS IMMEDIATELY AFTER INSPECTION.

THE ENHANCED CONDITION TABLE METHODOLOGY FOR VERIFICATION OF FAULT TOLERANT AND OTHER CRITICAL SOFTWARE

M. Hecht, K.S. Tso, and S. Hochhauser

SoHaR Incorporated
8500 Wilshire, Suite 1027, Beverly Hills, CA 90211
(213) 855-2595

1 Background and Motivation

Over the past decade, research on software fault tolerance has gained in importance as critical applications have become more software intensive. As work in fault tolerant software moves from research to application, verification will emerge as a critical issue. Fault tolerant systems have non-redundant components because the decision on a reconfiguration action must be taken at one point. These non-redundant components must be subject to an especially intensive verification. This paper and the accompanying viewgraphs describe a test-based methodology developed for this purpose.

Usual structural testing techniques such as path or branch testing are inadequate for such software, and other methods must be developed. The work described in this paper and the accompanying viewgraphs investigated enhancement of a verification methodology based on work performed by J. Goodenough and S. Gerhart based on condition tables [GOOD75]. Their method required testing not only all *paths* through the software, but all feasible combinations of *conditions*. The distinction between path testing and condition table testing is that in the former, the completion criterion is that all feasible paths are traversed at least once; in the latter, paths will be traversed many times with significantly different data. As a result of multiple traversals, it is more likely that coding errors such as incorrect conditions (e.g., IF A>B rather than if A $\geq$ B), incorrect operations (e.g., A=B+C instead of A=B*C), or missing branches (e.g., check for a value not being zero before dividing) will be detected.

The difficulties with the method is that it is excessively labor intensive. The effort to develop condition tables and test cases can be more than an order of magnitude greater than that required to develop the software undergoing test. Thus, it is impractical for full scale software development projects. Our objective was to enhance the methodology so that it applied to a realistic example: 1500 lines of code which comprise the kernel of a distributed fault tolerant system being developed for advanced nuclear reactor control under a contract to the Department of Energy. The enhancements that were developed include:

- Automated tools to generate the condition table
- An analysis format called the Test Case Enhancement Analysis (TCEA) for developing additional test cases which have functional, reliability, or safety

significance

- Implementation rules which simplify the generation of condition tables, test cases, and the creation of a test environment

2 Methodology Description

The steps for developing an enhanced condition table are:

1. Develop a condition table based on the code
2. Develop additional test case specifications by resolving "don't care" conditions into test cases which relate to specific functional, safety, or reliability concerns
3. Define test cases which satisfy the specifications developed in the first two steps
4. Create the test environment, run the tests and analyze the results.

Viewgraph 6 shows a simple code segment for managing a fault tolerant system consisting of two nodes, designated node1 and node2. If the outputs of the two nodes agree, then no further processing occurs. However, if there is disagreement between the nodes, then the first node is checked. If the check function returns a value of not OK (i.e., the node has failed the check), then the fail_node1 variable is set to TRUE. If the check function returns a value of OK (i.e., the node has passed the check), then the second node is checked. If the second node check value is OK, then a retry function is invoked. However, if the check indicates a failure, then the *fail_node2* variable is set to TRUE. If either of the two nodes have failed, then a reconfiguration function is invoked.

The resultant condition table is shown on the same viewgraph. The predicates from the 4 conditions are shown in the left hand column. The feasible combination of these predicates, called "rules", are shown in the 4 succeeding columns. The notation is as follows: y: condition set true; n: condition set false; (y) condition is necessarily true because of the state of other conditions; (n) condition is necessarily false; and -: irrelevant or "Don't Care". This format is an adaptation of the limited entry condition table first proposed by King [KING69].

The "Don't Care" conditions are the ones of concern in the condition table methodology. Under the Goodenough and Gerhart approach, the right-most column of the table would have to be decomposed into 8 (2^3) additional rules for which test cases would have to be written. The ECT approach instead requires that the analyst consider significant failure modes of the module using a format called the Test Case Enhancement Analysis (TCEA) shown in Viewgraph 7. The TCEA shows that only one additional case needs to be run: that the two nodes should agree *and* that they should be failed. This rule uncovers a significant flaw in the routine because such a case is not properly handled. Instead ordering a reconfiguration, the implementation of this module results in no action at all.

3 Tools Development

The tools development objective was addressed by the creation of two separate programs called *ECT* and *SEM*. The ECT program performs the following operations:

1. Lexical analysis that assembles terminal symbols (e.g., carriage return/line feed) and eliminates white space and comments

2. Syntactic analysis that parses C programs and detects grammatical errors based on the ANSI C language standard

3. Generation of a condition tree

4. Generation of a condition table from the condition tree.

The condition table generated by the ECT program is syntactically and structurally correct but contains many semantically infeasible rules. The second program, SEM, reduces the condition table generated by *ECT* based on an input file which contains the semantics of the C program.

Semantic information is input to the SEM program by means of an ASCII file using the following notation:

&	AND
|	OR
!	NOT
->	Implies that
~	Don't Care

For the code segment shown in viewgraph 6, the semantics are

!c2 | !c3 -> c4 (if the check on node 1 or node 2 is OK, then the node can not have failed).

The *ECT* and *SEM* tools decompose multiple conditions into unary conditions and generates tables based on the relationships of the multiple conditions. That is, a condition based on two predicates, e.g., if (a<b) and (c<d), becomes two conditions using the *ECT* and *SEM* programs; under the original approach, they would be regarded as a single condition. A second enhancement is that case and *while* statements are handled by the *ECT* and *SEM* programs whereas the original work dealt only with if-then-else constructs. *ECT* and *SEM* were implemented in C on a Sun 280 computer running SUNOS (UNIX 4.2 BSD), release 3.5. Figures 1 and 2 shows the top level structure of these programs; the parsing and lexical analysis portions were generated using the UNIX *yacc* and *lex* programs.

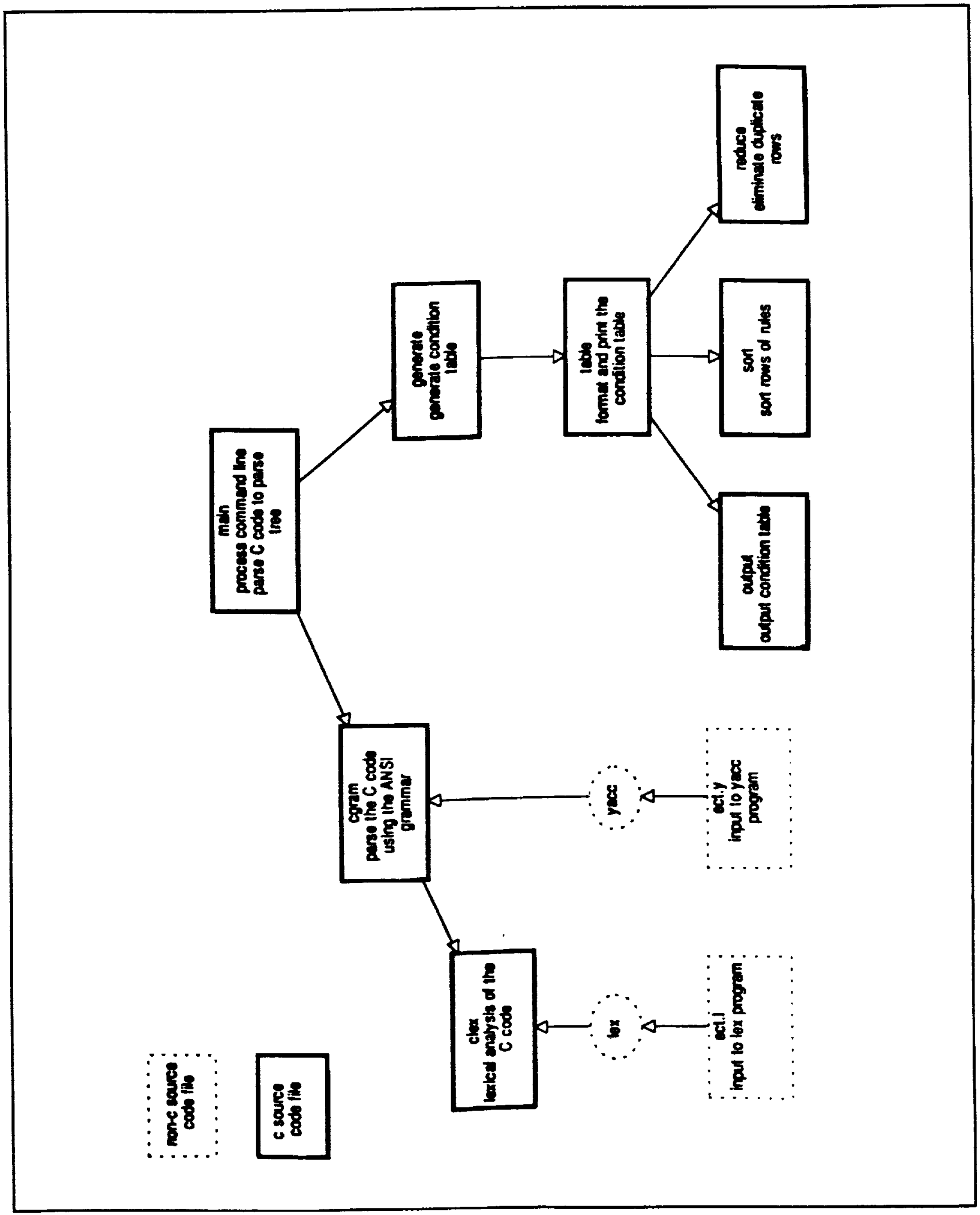


Figure 1. ECT Program Top Level Structure

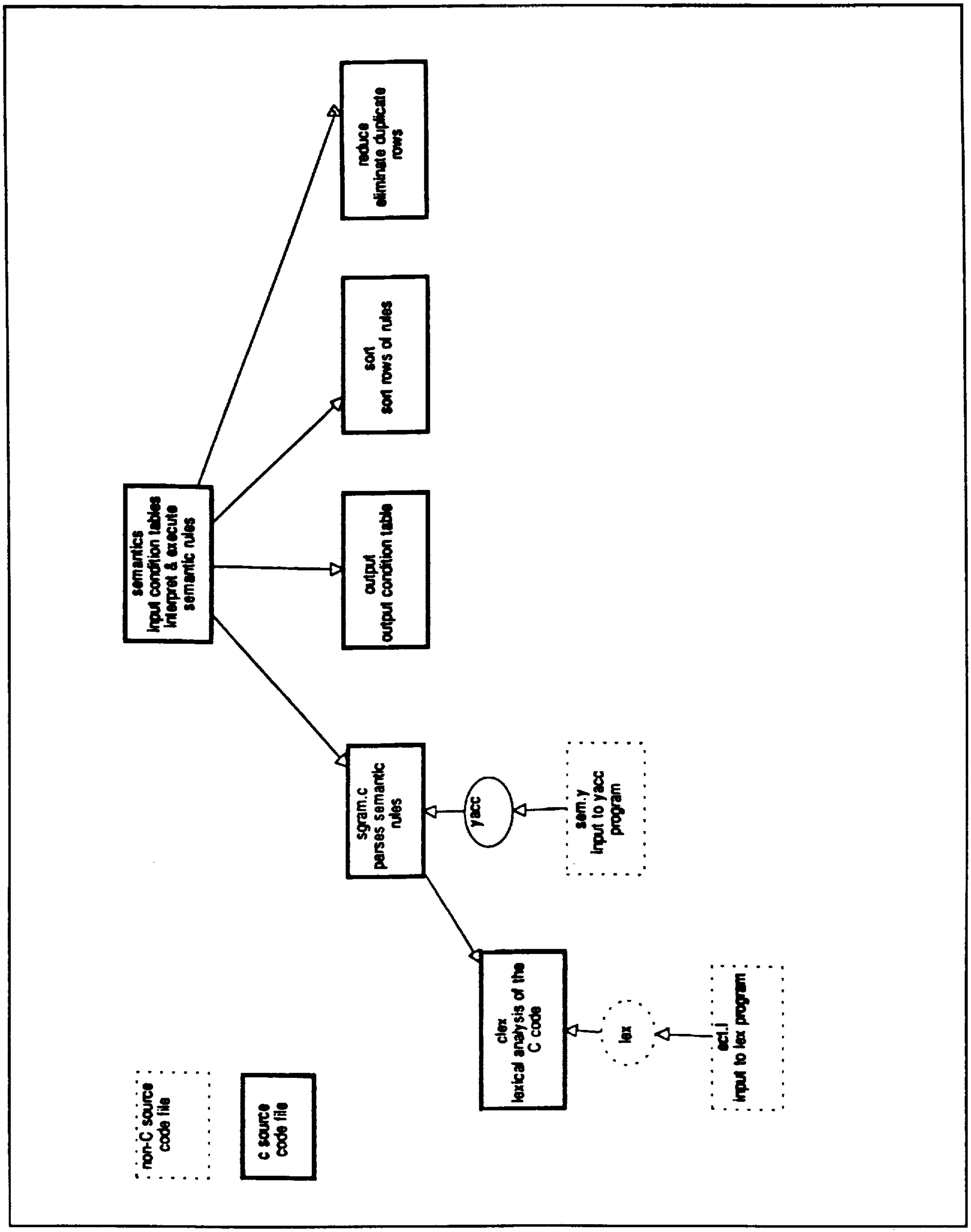


Figure 2. SEM Program top level structure

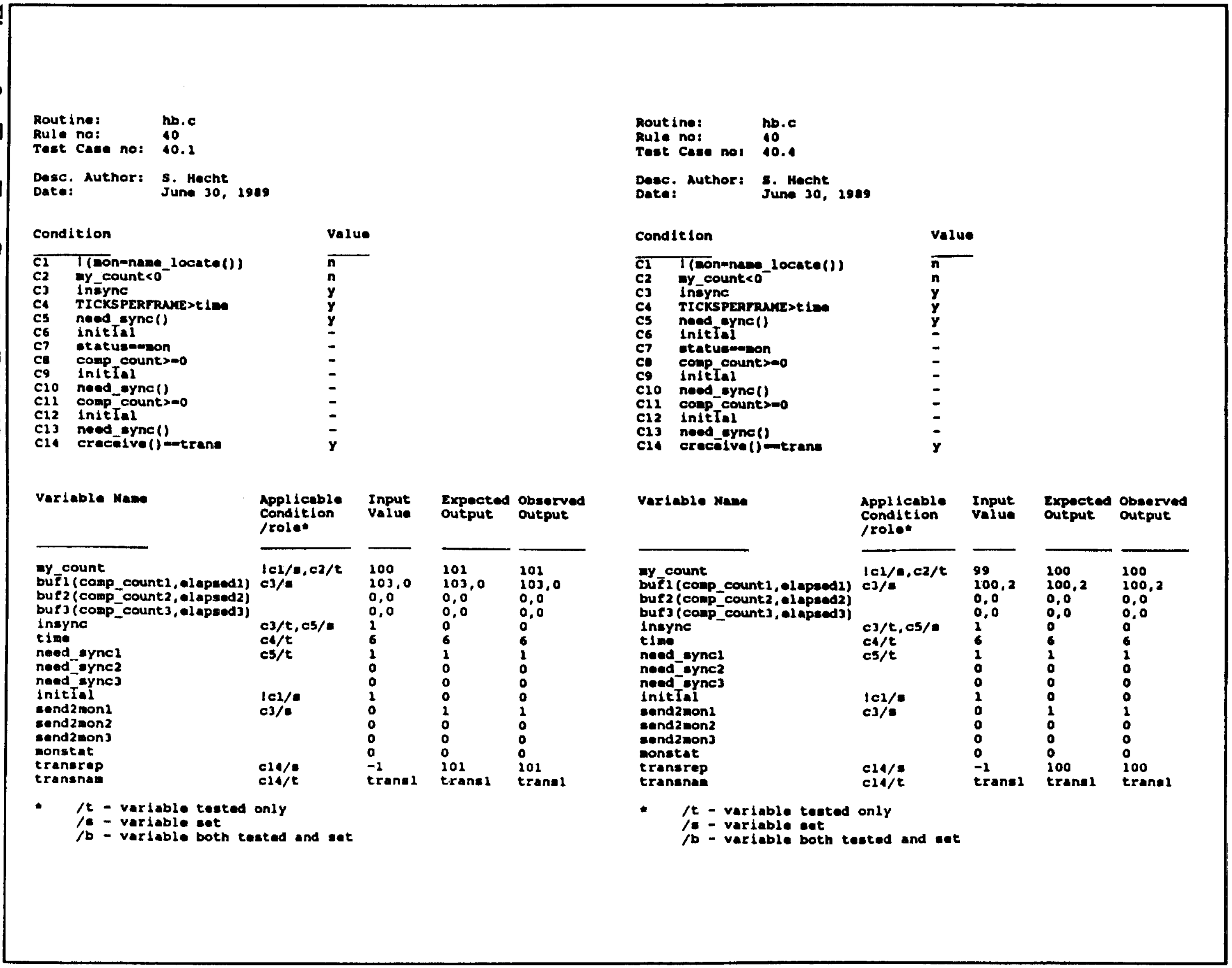
4 Feasibility Experiment

The feasibility experiment determined whether the ECT was indeed practical for a realistically sized critical software system by applying it to a fault tolerant distributed reactor control system being developed under another contract at SoHaR. The system, called the Extended Distributed Recovery Block, or EDRB, consists of approximately 2000 lines of ANSI standard C code; 1500 of which were sufficiently stable to warrant verification. A description of the EDRB system may be found in [HECH89].

Generation of condition tables was performed using the automated tools described above for all stable portions of the EDRB. The accompanying viewgraphs show an excerpt of code which is typical of the EDRB and the condition table generated by the ECT and SEM tools described in the previous section. The rules are listed in a horizontal format rather than the vertical format shown in the previous section because of the large number of rules. 700 rules were generated for the 30 modules. No difficulties were experienced in running the tools. The entire procedure took less than 1 hour, most of which was related to file transfers.

The next step was generation of the test cases from the rules defined by the condition table. Due to the 6-month schedule limitation of this Phase 1 SBIR research, only one of the tasks was chosen for testing. This task, called HEARTBEAT, was one of the most complicated in the EDRB, and is responsible for generating its own heartbeat, monitoring that of its shadow, and deciding whether to synchronize with its shadow or signal the system supervisor for a recovery action. A total of 109 test cases were developed for the routine. Multiple test cases were developed for some rules in order to test special values (i.e., values at or close to boundaries, discontinuities, etc.). Figure 3 shows two test cases that were generated for Rule 40. They differ in that the frame counts in one case are 99 and 100 whereas in the second they are 100 and 103. These two different sets of values test the limits of the behavior of the program in this branch. This approach exemplifies the combination of structural testing with other forms of testing in order to provide complete coverage. The fault found using the ECT which was not found using conventional inspection-based methods was due to a second test case on the 66th rule in the condition table.

The third step was generation of additional test cases that reflect concerns on the function of the module. A procedure was developed to generate such test cases and resulted in the formulation of the Test Case Enhancement Analysis, or TCEA, an excerpt of which is shown in Viewgraph 6. By using the TCEA, an additional several hundred rules were developed to resolve "Don't Cares" into y and *n* outcomes. However, the generation of these enhanced test cases was simplified by the fact that they are readily derived from existing test cases and their outcomes will be precisely identical to the existing test cases from which they were derived.Execution of the test cases required (1) modification of the source code, (2) a specially written test driver routine which reads in the test cases and outputs the results, and (3) stubs to substitute for subroutines and functions invoked by the unit under test. As will be discussed in the next chapter, creation of a unique test



Routine: hb.c
Rule no: 40
Test Case no: 40.1

Desc. Author: S. Hecht
Date: June 30, 1989

Condition		Value
C1	!(mon=name_locate())	n
C2	my_count<0	n
C3	insync	y
C4	TICKSPERFRAME>time	y
C5	need_sync()	y
C6	initial	-
C7	status==mon	-
C8	comp_count>=0	-
C9	initial	-
C10	need_sync()	-
C11	comp_count>=0	-
C12	initial	-
C13	need_sync()	-
C14	creceive()==trans	y

Variable Name	Applicable Condition /role*	Input Value	Expected Output	Observed Output
my_count	!c1/s,c2/t	100	101	101
buf1(comp_count1,elapsed1)	c3/s	103,0	103,0	103,0
buf2(comp_count2,elapsed2)		0,0	0,0	0,0
buf3(comp_count3,elapsed3)		0,0	0,0	0,0
insync	c3/t,c5/s	1	0	0
time	c4/t	6	6	6
need_sync1	c5/t	1	1	1
need_sync2		0	0	0
need_sync3		0	0	0
initial	!c1/s	1	0	0
send2mon1	c3/s	0	1	1
send2mon2		0	0	0
send2mon3		0	0	0
monstat		0	0	0
transrep	c14/s	-1	101	101
transnam	c14/t	transl	transl	transl

* /t - variable tested only
/s - variable set
/b - variable both tested and set

Routine: hb.c
Rule no: 40
Test Case no: 40.4

Desc. Author: S. Hecht
Date: June 30, 1989

Condition		Value
C1	!(mon=name_locate())	n
C2	my_count<0	n
C3	insync	y
C4	TICKSPERFRAME>time	y
C5	need_sync()	y
C6	initial	-
C7	status==mon	-
C8	comp_count>=0	-
C9	initial	-
C10	need_sync()	-
C11	comp_count>=0	-
C12	initial	-
C13	need_sync()	-
C14	creceive()==trans	y

Variable Name	Applicable Condition /role*	Input Value	Expected Output	Observed Output
my_count	!c1/s,c2/t	99	100	100
buf1(comp_count1,elapsed1)	c3/s	100,2	100,2	100,2
buf2(comp_count2,elapsed2)		0,0	0,0	0,0
buf3(comp_count3,elapsed3)		0,0	0,0	0,0
insync	c3/t,c5/s	1	0	0
time	c4/t	6	6	6
need_sync1	c5/t	1	1	1
need_sync2		0	0	0
need_sync3		0	0	0
initial	!c1/s	1	0	0
send2mon1	c3/s	0	1	1
send2mon2		0	0	0
send2mon3		0	0	0
monstat		0	0	0
transrep	c14/s	-1	100	100
transnam	c14/t	transl	transl	transl

* /t - variable tested only
/s - variable set
/b - variable both tested and set

Figure 3. Two Test Cases for Rule 40

ORIGINAL PAGE IS
OF POOR QUALITY

environment for each task is a labor intensive process which could be eliminated by an appropriate debugger capable of setting variables, tracing execution, and output results for off-line analysis.

5 Results and Discussion

Success in Use of Automated Tools

The most significant result of this work was that the automated tools *ECT* and *SEM* have eliminated a tedious and error-prone step in the verification process. Condition tables were generated for all 13 tasks (total of 30 main and subroutines written in ANSI C) in the EDRB node manager.

A related result is that automated tools provided a rapid unambiguous indication of excessive complexity. For example, one task in the EDRB had 20 conditions and 1050 rules. This result was in and of itself sufficient motivation for recoding of the module. Because these tasks were not contrived examples, it is reasonable to assume that the tools used to generate these condition tables can be used for any code which has less than 15 conditions.

Traceability of Test Cases and Results

The ECT methodology provides a complete and traceable test program. Traceability of all conditions in the code is provided through the automatically generated condition table. Safety and reliability analyses are traceable to the TCEA. The ECT and TCEA together form a test specification with unambiguous completion criteria. The test cases and results are in turn traceable to the test specification. This traceability makes the ECT a manageable process and facilitates IV&V, customer review, and regulatory agency review.

Fault Found Using the ECT Methodology

The ECT methodology uncovered a subtle fault in the HB routine synchronization algorithm. A special values test in a feasible path created a state in which a "stale" heartbeat count from the companion was greater than or equal to the local count. The local node synchronized on this stale count. Analysis of the anomaly showed that although the HB routine synchronizes on the frame *number*, it does not consider the age of the heartbeat message. This age is measured by a variable called *elapsed*, which counts the number of tenths of a frame that have elapsed since receipt of the most recent heartbeat. When the HB routine requests the companion heartbeat count from the monitor task, it also receives the elapsed value. If elapsed is greater than 2 ticks, then the program specification defines the nodes as no longer being in synchronization. However, in this case, a program variable called *insync* which indicates whether the nodes are in synchronization, was set to true even though no synchronization had actually occurred.

Earlier work with the ECT methodology also found a subtle fault [TAI87] in a fault detection and recovery section of a smaller module. A possible explanation of the nature of faults found by the methodology is that more obvious coding errors will have been detected during development and testing by the original software implementer. Thus, only errors in infrequently exercised paths with unusual values, i.e., "subtle" errors, will remain. However, previous research on data from the JPL Deep Space Network [MCCA87] has shown that such errors account for a disproportionate number of system failures. The structural aspect of the ECT methodology considers all code without regard to the functional aspects of the program. Because functionally oriented testing by the original developers will have occurred before the start of verification, only the subtle errors will remain.

Implementation Practices for Testability

A significant result of this research is the importance of designing and implementation of code to ensure testability. The following rules were found to be effective:

1. *No more than 12 conditions per module*: One measure of complexity of a module is the number of branches formed by if, while, else, and related conditions. As modules become more complex, they become more difficult to verify using either manual or automated approaches. Although the automated portions of the ECT methodology can handle modules of more than 20 conditions, they are difficult to understand and have an excessively large number of rules.

2. *Minimize setting of variable after using*: If the same variable is set and used several times for each execution or iteration of the unit under test, then it is difficult to conclusively evaluate its success because intermediate values are obliterated before the results are output. Following this rule allows the return values of *function* to be written to a record without an undue number of changes to the code and is particularly important for operating system calls and other black box modules. This method also reduces unintended interaction effects.

3. *Use parameters for subroutine calls, minimize use of global variables and pass subroutine arguments by value*: The principal advantage of information hiding for the purpose of the ECT is that it makes writing of a test driver easier and facilitates the creation of test cases. The other advantages of this rule are well known and would apply in general to high quality software.

4. *List parameters in the following order: (1) input parameters, (2) input/output parameters, and output parameters*: Although the input and output parameters must be separated in Ada, other languages such as C and FORTRAN, do not require explicit separation. The primary motivation for this rule is to facilitate testing. However, it also reduces the probability of inadvertently switching arguments or misunderstanding.

Following these rules reduced the number of changes by more than 60% in the modules tested. This reduction results in a more credible verification and also reduces the test effort.

Test Case Generation

The approach to test case generation in this phase of the research was to manually define the state of input values so that the path for each case was known a priori. The usual approach is to instrument the code and vary the input (many path testing programs vary the input randomly) within predefined ranges in the hopes of traversing most branches. When the automated testing is concluded, manual test cases are created only for the untraversed paths. The benefit of the a priori approach are:

1. A large reduction in the number of test cases that must be generated, stored, and evaluated

2. Explicit traceability of each test case to a rule, special values, and path

3. Easy creation of enhanced test cases generated from special values analysis and the Test Case Enhancement Analysis.

Test cases examine the states of internal variables as well as the input and output. Thus, testing requires the manipulation of internal variables, dynamically changing parameters, and other intrusive actions. The test environment can be entirely custom developed or can be built from existing tools within the operating system. In the QNX [QUAN88] operating system under which the EDRB runs, a test environment was specially written to read in test data, output results, and set values of internal variables and dummy variables substituted for operating system functions.

Different considerations would apply in more sophisticated software development environments. For example, the *dbx* tool in many implementations of UNIX 4.2 can set all internal variables and print out all output variables without the need for a driver routine; the test input can be read in through a script file and the output can be directed to the appropriate output file. Thus, the first and second principles would no longer apply. *dbx* also has some capabilities to control interaction with the operating system thereby reducing the need to replace calls to the operating system with test data variables.

Promising Areas for Additional Tools Development

The accompanying viewgraphs include an estimation the time requirements for the ECT given the current state of its development (i.e., the *ECT* and *SEM* tools). For a system the size of the EDRB, close to 1.5 technical staff years would be required. However, most of the effort is concentrated in three major tasks: generation of test cases, development of test environments, and resolution of don't cares and the creation of additional test cases.

An additional candidate for automation is the generation of semantic relations. Although not a particularly labor intensive task, it requires a detailed understanding of the semantics of the module and is prone to errors. Errors in the statement of semantic relations can in turn invalidate the rest of the ECT effort.

Therefore, the most promising areas four tools development are:

1. *Semantic Analyzers:* The *SEM* program requires a semantic data input file which is manually created. Experience in creating such files has shown that the process is subject to analyst error. A semantic analyzer can generate such files automatically.

2. *Test Case Generator*: Test case generation requires identifying those input and externally set variables that satisfy (or do not satisfy) each condition and the values that should be assigned to these variables in order to execute the path specified by each rule. This identification is a matter of tracing how variables are set and used and may be amenable to automation using an existing static analyzer. This tool would find each condition in the code, trace variables associated with these conditions to their inputs, determine what ranges input values should be used to satisfy the specifications imposed by the rule, prepare a test case input file, and print the test case.

3. *Debugging Script Generator*: The work needed to generate a test environment can be largely reduced if batch oriented debugging tools such as *dbx* are used. The importance of the batch orientation is that test cases can be written off-line and input as files to the debugger, and debugger output can likewise be examined either manually or automatically.

4. *Support for Generation of Additional Test Cases*: Resolution of *Don't Cares* and the generation of additional test cases requires understanding of system-level concerns and association of these concerns with variables and control flow of the unit under test. By definition, the process can not be automated; otherwise the designed could be analyzed and the concerns would be resolved. However, providing cross references to variables and control flows which reduce the repetitive labor involved in performing this analysis, will result in a significant reduction of labor and in more thorough and uniformly high quality analyses.

6 Conclusions

The result of this work was that it was possible to analyze a large section of code which shares many characteristics in common with future real time distributed control systems that will be implemented in the next generation of aircraft and space vehicles. The results further showed that traceable test case specifications are generated, that unambiguous completion criteria can be established, and that automated tools can be successfully used.

Additional work is necessary to reduce time and resource requirements by the development of appropriate tools. The benefits of such tools are exemplified by generation of the basic condition table which was previously an error prone and labor intensive task. With the *ECT* and *SEM* tools, this step has been reduced to a negligible portion of the total effort.

7 Acknowledgements

This work was sponsored by the NASA Langley Research Center under contract NAS1-18811. The authors wish to acknowledge the interest and support of Mr. Carlos Liciega, the technical monitor, as well as Ms. Susan Voigt, who provided the motivation which resulted in this research.

References

KING69 P. King, "The interpretation of limited entry decision table format and relationships among conditions", Computing Journal, Vol 12, p. 320, November, 1969

GOOD75 J. Goodenough and S. Gerhart, "Toward a Theory of Test Data Selection", *IEEE Transactions on Software Engineering*, Vol. SE-1 No. 2, June, 1975, p. 156

MCCA87 J. M. McCall, et. al., *Methodology for Software Reliability Prediction*, Rome Air Development Center, RADC-TR-87-171, November, 1987

QUAN88 Quantum Software Inc., *QNX Operating System Reference Manual,* available from Quantum Software, Kanata, Ontario, Canada, 1988

TAI87 A. Tai, M. Hecht, and H. Hecht, "A New Method for the Verification of Fault Tolerant Software", *Proc. EASCON '87*, IEEE Catalog No. 87CH 2491-9, November, 1987, p. 53

VIEWGRAPH MATERIALS

FOR THE

M. HECHT PRESENTATION

Enhanced Condition Tables

for Verification of Fault Tolerant Software

Contract NAS1-18811

presented to

Fourteenth Annual Software Engineering Workshop

NASA Goddard Space Flight Center
Greenbelt, MD

by

SoHaR Incorporated
Los Angeles, CA

November, 1989

BACKGROUND

Method based on work done by J. Goodenough and S. Gerhart in the 1970s

Showed that path testing was not adequate for common error types

Omission of paths (failure to check for divide by 0)
Wrong conditions (IF A> 0 instead of IF >= 0)
Wrong or missing actions on a path (a*b instead of a+b)

Demonstrated condition table as a method of generating more effective test sets

Testing of all feasible combinations of conditions poses a much greater likelihood of revealing the fault because of multiple passes through each path with variations that would reveal wrong conditions or missing actions

A methodology for effective test case generation is essential for critical applications. The general approach appeared to be the most promising for fault tolerant software being developed at SoHaR

BACKGROUND (continued)

The method was impractical for larger programs

- In a realistic program, generation of condition tables is a time consuming and error prone process
  - A combinatorial explosion resulting in thousands of rules would occur even in a small module if all feasible combinations of conditions are considered
- The effort for generating test cases would exceed the coding effort by an order of magnitude
- "case" and "while" statements were not considered in the original methodology

We sought to alleviate these difficulties

- Develop tools to generate condition tables automatically
- Develop methodologies to reduce the number of rules
- Identify coding practices that simplify generation of test cases
- Identify additional functions to reduce the effort in test case generation

ECT METHODOLOGY

Generate CTs (performed using two programs ECT and SEM)

Generate additional rules to resolve "Don't Cares" based on functional requirements, special values, and results of safety and reliability Analyses

Define Test Cases (including expected results)

Create the test environment (drivers and stubs)

Run test cases and analyze results

ECT METHODOLOGY (CONTINUED)

SAMPLE PROGRAM

```
if (! consensus(node1, node2)) {
    if (check(node1) == OK) {
        if (check(node2) == OK) {
            retry();
        } else {
            fail_node2 = true;
        }
    } else {
        fail_node1 = true;
    }
    if (fail_node1 || fail_node2)
        reconfig();
}
```

RESULTANT CONDITION TABLE

c1: !consensus(node1, node2)	y	y	y	n
c2: check(node1)	y	y	n	-
c3: check(node2)	y	n	-	-
c4: fail_node1 || fail_node2	(n)	(y)	(y)	-

ECT METHODOLOGY (CONTINUED)

Test Case Enhancement Analysis

Rule	Condition	Variable	Severity	Test Case
4	c2	check (node1)	Severe	Set return values of check to show that node1 and node2 have failed.
	c3	check (node2)		Success Criterion: reconfig() is invoked (even though answers of both nodes agree)
	c4	fail_node1 fail_node2		Failure Criterion: reconfig() not invoked

ECT METHODOLOGY (CONTINUED)

Programs for Automatically Generation Condition Tables

ECT

Parses C code and extracts the conditional expressions from "if", "switch", "while", and "for" statements

Detects grammatical errors (deviation from ANSI C standard)

Decomposes multiple conditions into unary conditions and generates tables based on the relationships of the multiple conditions

Generates condition table without consideration of program semantics

SEM

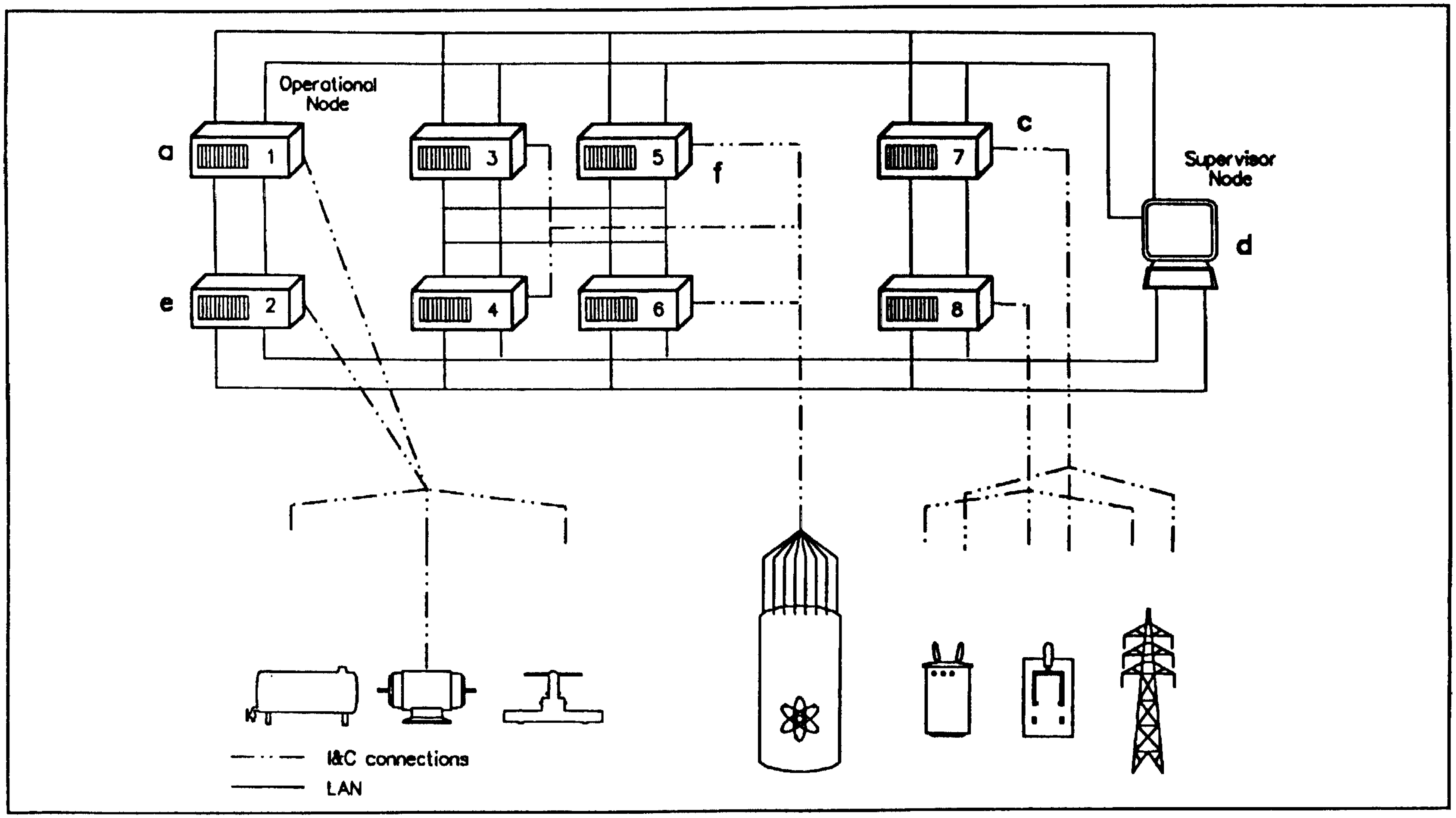
Reduces size of condition table by consideration of program semantics

Current version requires user to enter semantics as a text file

Future versions will elimante manual entry (manual entry can cause perpetuation of erroneous assumptions leading to overlooking ofimplementation errors)

FEASIBILITY EXPERIMENT

Target System Description



FEASIBILITY EXPERIMENT (continued)

Node Manager System Functions

Node Manager Generates Periodic Status Messages (called a "Heartbeat")

Determines Operational Status of Companion (called "Health")

Determines Role (Active or Shadow)

Determines Version of Software to Run (Primary or Alternate)

Determines Acceptability of Results

HB Module Description

Responsible for initializing system during startup

Driven by periodic signal generated internally (every 0.5 seconds)

Checks companion heartbeat and resynchronizes if necessary

Initiates reboot process on companion node if no heartbeat arrives (additional consent and processing required to complete)

```
hb.c        Sun May 14 10:27:29 1989        1

/*
 * This program task generates a heartbeat signal every frame, which it
 * replies to the TRANS task if it is send blocked on this task.  TRANS is a
 * spawned concurrent task, created by this task to transmit the heartbeat
 * count to the companion node and the supervisor.  Typing a 'q' (or 'Q') at
 * any time will exit the program.  The frame size is based on the constant
 * TICKSPERFRAME (specified in hb.h).  This task tries to maintain
 * synchronization with the companion node via the monitor task on this node
 * (which receives heartbeats from the companion node).
 */
#include <magic.h>
#include <stdio.h>
#include <dev.h>
#include <timer.h>
#include "hb.h"

main()
{
    int             my_count = 0;        * heartbeat count */
    int             comp_count = 0;     /* companion heartbeat count */
    int             elapsed = 0;/* ticks elapsed since comp_count was rec'd */
    unsigned        trans;
    int             i;
    int             status;
    unsigned        mon;
    int             insync = FALSE;     /* TRUE if this task is synchronized
                                         * with companion */
    int             initial = TRUE;     /* TRUE if this is the first
                                         * heartbeat */
    char            buf[4], countbuf[2], mynode[3];
    FILE           *fp_kbd;
    unsigned        tick1, tick2, time;
    extern unsigned Exc_permit[2], Exc_allow[2];


    set_priority(4);            /* set priority higher than all applications */
    My_priority = 4;

    exc_handler(0, 0, 0);
    Exc_permit[0] = 0xffff;     /* ignore system exceptions on this task */
    Exc_allow[0] = 0x0;
    name_attach("hb", My_nid);

    trans = createl(0, 0, -1, 0, 2, My_priority, 0, 0, 0, "trans", 0);
    fp_kbd = stdin;
    set_option(fp_kbd, get_option(fp_kbd) & ~(EDIT | ECHO));

    while (!(mon = name_locate("mon", My_nid, 5)));     /* wait for monitor task */
    set_timer(TIMER_WAKEUP, RELATIVE, 20);      /* wait for final task
                                                 * creations */

    tick1 = get_ticks();

    /* loop forever */
    for (;;) {
        my count += 1;
        if (my_count < 0)
            my count = 0;       /* cycle around */
        countbuf[0] = my_count & 0xff;
        countbuf[1] = my_count >> 8;

        check_exit(fp_kbd, trans);      /* check keyboard input (to exit
                                         * program) */

        /*
         * Determine amount of time (# of ticks) elapsed since the last
         * heartbeat was generated, so that it can be subtracted from
         * TICKSPERFRAME to establish the desired "wait time" for the next
         * heartbeat.
         */
        tick2 = get_ticks();
        time = (tick2 < tick1) ? (0xffff - tick1) + tick2 : tick2 - tick1;
        tick1 += TICKSPERFRAME;

        if (insync) {
            if (TICKSPERFRAME > time)
                set_timer(TIMER_WAKEUP, RELATIVE, TICKSPERFRAME - time);
            else
                tick1 = get_ticks();
            send(mon, "r", buf, 1);     /* get most recent comp hb from
                                         * monitor */
            comp_count = (buf[1] << 8) + (buf[0] & 0xff);
            elapsed = (buf[3] << 8) + (buf[2] & 0xff);
            if (need_sync(my_count, comp_count, elapsed))
                insync = FALSE;
            else
                insync = TRUE;
        } else {                    /* !insync */
            if (initial)
                set_timer(TIMER_FORCE_READY, RELATIVE, 3 * TICKSPERFRAME);
            else
                set_timer(TIMER_FORCE_READY, RELATIVE, TICKSPERFRAME);
            status = send(mon, "n", buf, 1);    /* wait for next comp hb from
                                                 * monitor */
            set_timer(TIMER_CANCEL, RELATIVE);
            comp_count = (buf[1] << 8) + (buf[0] & 0xff);
            elapsed = 0;
            tick1 = get_ticks();/* reset hb frame */
            if ((status == mon) && (comp_count >= 0) &&
                (initial || need_sync(my_count, comp_count, elapsed))) {
                my_count = comp_count;
                countbuf[0] = my_count & 0xff;
                countbuf[1] = my_count >> 8;
            } else {
                send(mon, "r", buf, 1); /* get most recent comp hb from
                                         * monitor */
                comp_count = (buf[1] << 8) + (buf[0] & 0xff);
                elapsed = (buf[3] << 8) + (buf[2] & 0xff);
                if ((comp_count >= 0) &&
                    (initial || need_sync(my_count, comp_count, elapsed))) {
                    my_count = comp_count;
                    countbuf[0] = my_count & 0xff;
                    countbuf[1] = my_count >> 8;
                }
            }
            insync = TRUE;
        }                           /* ! insync */
        initial = FALSE;

        if (creceive(trans, buf, 0) == trans)
            reply(trans, countbuf, 2);
    }                               /* for(;;) */
}

need_sync(my_count, comp_count, elapsed)
    int             my_count;   /* My node's current heartbeat count */
    int             comp_count; /* Companion's current heartbeat count */
    int             elapsed;    /* Ticks elapsed since comp_count was lst
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
hb.out          Mon May 29 20:50:26 1989          1

   Condition Table (with semantics knowledge) for function "main"

                                                                                Page 1

                                                 c1   c2   c3   c4   c5   c6   c7   c8   c9   c10  c11  c12  c13  c14

c1:  !(mon=name_locate())                   1    y    -    -    -    -    -    -    -    -    -    -    -    -    -
c2:  my_count<0                             2    n    y    y    y    y    -    -    -    -    -    -    -    -    y
c3:  insync                                 3    n    y    y    y    y    -    -    -    -    -    -    -    -    n
c4:  TICKSPERFRAME>time                     4    n    y    y    y    n    -    -    -    -    -    -    -    -    y
c5:  need_sync()                            5    n    y    y    y    n    -    -    -    -    -    -    -    -    n
c6:  initial                                6    n    y    y    n    y    -    -    -    -    -    -    -    -    y
c7:  (status==mon)                          7    n    y    y    n    y    -    -    -    -    -    -    -    -    n
c8:  (comp_count>=0)                        8    n    y    y    n    n    -    -    -    -    -    -    -    -    y
c9:  (initial                               9    n    y    y    n    n    -    -    -    -    -    -    -    -    n
c10: need_sync())                          10    n    y    n    -    -    y    y    y    y    -    -   (y)   -    y
c11: (comp_count>=0)                       11    n    y    n    -    -    y    y    y    y    -    -   (y)   -    n
c12: (initial                              12    n    y    n    -    -    y    y    y    n    y    -    -    -    y
c13: need_sync())                          13    n    y    n    -    -    y    y    y    n    y    -    -    -    n
c14: creceive()==trans                     14    n    y    n    -    -    y    y    y    n    n    y    y    -    y
                                           15    n    y    n    -    -    y    y    y    n    n    y    y    -    n
                                           16    n    y    n    -    -    y    y    y    n    n    y    n    y    y
                                           17    n    y    n    -    -    y    y    y    n    n    y    n    y    n
                                           18    n    y    n    -    -    y    y    y    n    n    y    n    n    y
                                           19    n    y    n    -    -    y    y    y    n    n    y    n    n    n
                                           20    n    y    n    -    -    y    y    y    n    n    n    -    -    y
                                           21    n    y    n    -    -    y    y    y    n    n    n    -    -    n
                                           22    n    y    n    -    -    y    y    n    -    -    y    y    -    y
                                           23    n    y    n    -    -    y    y    n    -    -    y    y    -    n
                                           24    n    y    n    -    -    y    y    n    -    -    y    n    y    y
                                           25    n    y    n    -    -    y    y    n    -    -    y    n    y    n
                                           26    n    y    n    -    -    y    y    n    -    -    y    n    n    y
                                           27    n    y    n    -    -    y    y    n    -    -    y    n    n    n
                                           28    n    y    n    -    -    y    y    n    -    -    n    -    -    y
                                           29    n    y    n    -    -    y    y    n    -    -    n    -    -    n
                                           30    n    y    n    -    -    y    n    -    -    -    y    y    -    y
                                           31    n    y    n    -    -    y    n    -    -    -    y    y    -    n
                                           32    n    y    n    -    -    y    n    -    -    -    y    n    y    y
                                           33    n    y    n    -    -    y    n    -    -    -    y    n    y    n
                                           34    n    y    n    -    -    y    n    -    -    -    y    n    n    y
                                           35    n    y    n    -    -    y    n    -    -    -    y    n    n    n
                                           36    n    y    n    -    -    y    n    -    -    -    n    -    -    y
                                           37    n    y    n    -    -    y    n    -    -    -    n    -    -    n
                                           38    n    y    n    -    -    n    y    y    y    -    -   (y)   -    y
                                           39    n    y    n    -    -    n    y    y    y    -    -   (y)   -    n
                                           40    n    y    n    -    -    n    y    y    n    y    -    -    -    y
                                           41    n    y    n    -    -    n    y    y    n    y    -    -    -    n
                                           42    n    y    n    -    -    n    y    y    n    n    y    y    -    y
                                           43    n    y    n    -    -    n    y    y    n    n    y    y    -    n
                                           44    n    y    n    -    -    n    y    y    n    n    y    n    y    y
                                           45    n    y    n    -    -    n    y    y    n    n    y    n    y    n
                                           46    n    y    n    -    -    n    y    y    n    n    y    n    n    y
                                           47    n    y    n    -    -    n    y    y    n    n    y    n    n    n
                                           48    n    y    n    -    -    n    y    y    n    n    n    -    -    y
                                           49    n    y    n    -    -    n    y    y    n    n    n    -    -    n
                                           50    n    y    n    -    -    n    y    n    -    -    y    y    -    y
```

ORIGINAL PAGE IS
OF POOR QUALITY

Routine: hb.c
Rule no: 2
Test Case no: 2.1

Desc. Author: S. Hecht
Date: June 8, 1989

Condition		Value
C1	!(mon=name_locate())	n
C2	my_count<0	y
C3	insync	y
C4	TICKSPERFRAME>time	y
C5	need_sync()	y
C6	initial	-
C7	status==mon	-
C8	comp_count>=0	-
C9	initial	-
C10	need_sync()	-
C11	comp_count>=0	-
C12	initial	-
C13	need_sync()	-
C14	creceive()==trans	y

Variable Name	Applicable Condition /role*	Input Value	Expected Output	Observed Output
my_count	!c1/s,c2/b	32767	0	0
buf1(comp_count1,elapsed1)	c3/s	5,0	5,0	5,0
buf2(comp_count2,elapsed2)		0,0	0,0	0,0
buf3(comp_count3,elapsed3)		0,0	0,0	0,0
insync	c3/t,c5/s	1	0	0
time	c4/t	6	6	6
need_sync1	c5/t	1	1	1
need_sync2		0	0	0
need_sync3		0	0	0
initial	!c1/s	1	0	0
send2mon1	c3/s	0	1	1
send2mon2		0	0	0
send2mon3		0	0	0
monstat		0	0	0
transrep	c14/s	-1	0	0
transnam	c14/t	trans1	trans1	trans1

* /t - variable tested only
/s - variable set
/b - variable both tested and set

RESULTS

1 Fault found using ECT

A "stale" companion heartbeat greater than recent local heartbeat could result in decision not to synchronize when synchronization was necessary.

Example of omission of paths error classification from Goodenough and Gerhart.

Rare case would not have been found during path, functional or integration testing.

Typical of errors that cause a disproportionate number of failures once system becomes operational

14 Faults found using other methods (functional and integration testing): None of these faults were related to the module that underwent ECT testing

Failures in network communications caused by creation of too many virtual circuits during device timeouts (resulted in overflow)

Task Deadlock

Intermittent network driver failures caused by race conditions between interrupts and setting of interrupt mask bits (i.e., poor device driver design)

RESULTS (continued)

Labor Effort Estimate

Task	Labor Effort	Requirement
Generation of Semantic Relations	8 hr/ mode	104 hr
Generation of TCEA	10 hr/task	130 hr
Generation of Condition Table	1 hr/run	1 hr
Generation of Feasible Rule Test Cases	2 hr/rule	1400 hr
Generation of Special Values	10 min/test case	12 hr
Resolution of Don't Cares and generation of additional test cases	1 hr/rule	350 hr
Generation of test environment (drivers, stubs, and routine modification)	80 hrs/task	1040 hr
Running of test cases	1 hr/run	13 hr
TOTAL		2946 hr

RESULTS (continued)

Integration of functional with structural testing provides benefits for traceability

Test cases can be directly traced to both specific paths and to specific functional or safety concerns

Less ambiguous completion criteria

Tools are effective

ECT and SEM tools were run on entire software system.

Minor bugs were encountered and corrected

A labor intensive and error prone part of the process was automated

Coding Practices Can Reduce Effort

- No more than 12 conditions per module
- Minimize setting of variables after using
- Use only parameters passed by value for subroutine calls

DISCUSSION

ECT in the context of a V&V effort

- FMEA, Fault Tree Analysis or other semi-quantitative analysis provides input to ECT and validation testing
- ECT is effective for module or unit testing
  - Subtle faults were detected that would not be found by other methods
  - Traceable records are provided to facilitate subsequent IV&V, qualification, certification, or customer acceptance
- Functional testing for end-to-end validation
  - Integrated system is necessary to test for timing, interface, and response time
  - Early and continuous testing can be facilitated if development system is similar to target system

DISCUSSION

Additional Tools

Coding style checkers

Flag violations of implementation for testability rules

Semantic Analyzers

Current manual generation of semantics is an error prone activity that can perpetuate erroneous assumptions

Test Case Generation Tools

Aids required to automate most time consuming part of the process

Can use output of static analyzers to associate inputs with variables under test, interact with the user to determine expected outcomes

Automation does not relieve testers from functional knowledge of the code and of safety and reliability concerns

CONCLUSIONS

ECT methodology can be used for verification of actual real time control system

Intensive verification is not sufficient; interface testing and end-to-end validation is also necessary as part of a full V&V program

Additional tools are necessary to reduce the labor and time required to create test environments and to generate test cases

When such tools are developed, the ECT will be a thorough, traceable, and effective means of testing at the unit level

APPENDIX A — ATTENDEES

FOURTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP ATTENDEES

ADAMS, KIRK	COMPUTER SCIENCES CORP.
ADAMS, NEIL	BENDIX FIELD ENGINEERING CORP.
AGRESTI, BILL W.	THE MITRE CORP.
AIKENS, STEPHEN D.	DEPT. OF DEFENSE
ALANEN, JACK	SOHAR, INC.
AMBROSE, LESLIE	THE MITRE CORP.
AMMANN, PAUL E.	GEORGE MASON UNIVERSITY
ANDERSEN, BILL	DEPT. OF DEFENSE
ANDERSON, FRANCES	STANFORD TELECOMMUNICATIONS, INC.
ANGIER, BRUCE	INSTITUTE FOR DEFENSE ANALYSIS
ARMSTRONG, MARY	IIT RESEARCH INSTITUTE
ARMSTRONG, ROSE	MOUNTAINET, INC.
ARNOLD, ROBERT S.	SOFTWARE PRODUCTIVITY CONSORTIUM
ASHTON, ANNETTE	NAVAL SURFACE WEAPONS CENTER
ASTILL, PATRICIA	CENTEL FEDERAL SERVICES
ATKINS, EARL	ELECTRONIC WARFARE ASSOCIATION
AZUMA, KENNETH I.	FORD AEROSPACE CO.
BACHMAN, SCOTT	DEPT. OF DEFENSE
BARDIN, BRYCE M.	HUGHES AIRCRAFT CO.
BARKSDALE, JOE	NASA/GSFC
BARNES, BRUCE H.	NATIONAL SCIENCE FOUNDATION
BARNES, DAVID	UNISYS CORP.
BARSKY, JERRY	BENDIX FIELD ENGINEERING CORP.
BASILI, VIC	UNIVERSITY OF MARYLAND
BAYNES, PERCY	VITRO CORP.
BEALL, SHELLEY	SOCIAL SECURITY ADMINISTRATION
BENITEZ, MEG	DEPT. OF DEFENSE
BEWTRA, MANJU	CTA, INC.
BIOW, CHRISTOPHER	DEFENSE COMMUNICATIONS AGENCY
BLAGMON, LOWELL E.	NAVAL CENTER FOR COST ANALYSIS
BLUM, BRUCE I.	JOHNS HOPKINS UNIVERSITY
BLUMBERG, MAURICE	IBM
BOND, JACK	DEPT. OF DEFENSE
BOND, PAUL	SAIC
BOOTH, ERIC	COMPUTER SCIENCES CORP.
BOURNE, WILLIAM	AMERICAN SYSTEMS CORP.
BOYCE, MARY-ANN	RMS TECHNOLOGIES, INC.
BRAUN, CHRIS	CONTEL TECHNOLOGY CENTER
BREDESON, RICHARD W.	OMITRON, INC.
BRIAND, LIONEL	UNIVERSITY OF MARYLAND
BRINKER, ELISABETH	NASA/GSFC
BRISTOW, JOHN	NASA/GSFC
BROPHY, CAROLYN	NAVAL RESEARCH LAB
BROWN, HARROLD E.	NASA/MSFC
BROWN, MARTY	COMPUTER SCIENCES CORP.
BUCHANAN, GEORGE A.	IIT RESEARCH INSTITUTE
BUCKLEY, JOE	COMPUTER SCIENCES CORP.
BUHLER, MELANIE	COMPUTER SCIENCES CORP.
BURCAK, THOMAS M.	PLANNING RESEARCH CORP.
BURLEY, RICK	NASA/GSFC
BUSBY, MARY B.	IBM
BUSH, MARILYN	NASA/JPL

FOURTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP ATTENDEES

CAKE, SPENCER C.	HQ USAF/SCTT
CALDIERA, GIANLUIGI	SOFTSIEL
CANTONE, GIOVANNI	UNIVERSITY OF MARYLAND
CARD, DAVE	COMPUTER SCIENCES CORP.
CARDENAS, SERGIO	UNIVERSITY OF MARYLAND
CARLISLE, CANDACE	NASA/GSFC
CARMODY, CORA	PLANNING RESEARCH CORP.
CARPENTER, MARIBETH B.	CARNEGIE MELLON UNIVERSITY
CARRIO, MIGUEL	TELEDYNE BROWN ENGINEERING
CASASANTA, RALPH	COMPUTER SCIENCES CORP.
CAUSEY, MICHAEL A.	COMPUTER SCIENCES CORP.
CERNOSEK, GARY J.	MCDONNELL DOUGLAS SPACE SYSTEMS CO
CHASSON, MARGARET C.	IBM
CHEDGEY, CHRIS	SPAR AEROSPACE CO.
CHMURA, LOUIS J.	NAVAL RESEARCH LAB
CHUNG, ANDREW	FAA TECHNICAL CENTER
CHURCH, VIC	COMPUTER SCIENCES CORP.
CISNEY, LEE	NASA/GSFC
COBARRUVIAS, JOHN R.	NASA/JSC
COHEN, SARA	GENERAL ELECTRIC CORP.
COLEMAN, MONTE	DEPT. OF THE ARMY
COOK, JOHN F.	NASA/GSFC
CORBIN, REGINA	SOCIAL SECURITY ADMINISTRATION
COTNOIR, DONNA	COMPUTER SCIENCES CORP.
COUCHOUD, CARL B.	SOCIAL SECURITY ADMINISTRATION
COVER, DONNA	COMPUTER SCIENCES CORP.
CRAWFORD, STEW	
CREASY, PHIL	MCDONNELL DOUGLAS ASTRONAUTICS CO.
CREECY, RODNEY	HUGHES AIRCRAFT CO.
CREEGAN, JIM	FORD AEROSPACE CO.
CREPS, DICK	UNISYS CORP.
CROKER, JOHN	LISAN CORP.
D'AGOSTINO, JEFF	OAO CORP.
DAKU, WALTER	VITRO CORP.
DANGERFIELD, JOSEPH W.	TELESOFT
DAS, PRASANTA	THE ANALYTIC SCIENCES CORP.
DECKER, WILLIAM	COMPUTER SCIENCES CORP.
DEGRAFF, GEORGE	GRUMMAN
DEMAIO, LOUIS	NASA/GSFC
DEUTSCH, MICHAEL S.	HUGHES AIRCRAFT CO.
DEWBRE, DOYLE	DEPT. OF DEFENSE
DIGNAN, DAVID M.	DEPT. OF DEFENSE
DODD, JOHN C.	COMPUTER SCIENCES CORP.
DOUGLAS, FRANK J.	SOFTRAN,INC.
DUNCAN, SCOTT P.	BELL COMMUNICATIONS RESEARCH, INC.
DUNN, NEPOLIA	COMPUTER SCIENCES CORP.
DUQUETTE, RICHARD	DEPT. OF LABOR
DUREK, TOM	TRW
DUTTINE, VALERIE	NASA/GSFC
DUVALL, LORRAINE	DUVALL COMPUTER TECHNOLOGIES,INC.
DYER, MICHAEL	IBM

FOURTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP ATTENDEES

EARL, MICHAEL	INTERMETRICS, INC.
EDWARDS, JOHN	IIT RESEARCH INSTITUTE
EDWARDS, STEPHEN G	NASA/GSFC
EGLITIS, JOHN	LOGICON, INC.
ELLIOT, MATTHEW	NASA/STX
ELLIOTT, DEAN F	SWALES & ASSOCIATES INC.
ELLIS, WALTER	IBM
EMEIGH, MICHAEL	LOGICON, INC.
EMERSON, CURTIS	NASA/GSFC
EMERY, RICHARD	VITRO CORP.
EPSTEIN, WILLIAM	IBM
ERB, DONA M	THE MITRE CORP.
ESHLEMAN, LAURA	DEPT. OF DEFENSE
ESKER, LINDA	COMPUTER SCIENCES CORP.
EUSTICE, ANN	IIT RESEARCH INSTITUTE
EVANCO, WILLIAM	THE MITRE CORP.
FARR, BILL	NAVAL SURFACE WEAPONS CENTER
FEERRAR, WALLACE	THE MITRE CORP.
FERNANDEZ, AL	COMPUTER SCIENCES CORP.
FERRY, DAN	COMPUTER SCIENCES CORP.
FINK, MARY LOUISE A	EPA
FISHKIND, STAN	NASA/HEADQUARTERS
FONG, GEORGE	IIT RESEARCH INSTITUTE
FORSYTHE, RON	NASA/WALLOPS FLIGHT FACILITY
FOURROUX, KATHY	TELEDYNE BROWN ENGINEERING
FOUSER, THOMAS J	JET PROPULSION LAB
GACUK, PETER	SPAR AEROSPACE CO.
GAFFKE, WILLIAM E	PROJECT ENGINEERING, INC.
GAFFNEY, JOHN	SOFTWARE PRODUCTIVITY CONSORTIUM
GAITHER, MELISSA	CRMI
GALLAGHER, BARBARA	DEPT. OF DEFENSE
GARCIA, ENRIQUE A	JET PROPULSION LAB
GARRETT, TOM	IRS
GARY, ALAN V	TELEDYNE BROWN ENGINEERING
GELPERIN, DAVID	SOFTWARE QUALITY ENGINEERING
GIESER, JIM	VITRO CORP.
GILSTRAP, LEWEY	COMPUTER SCIENCES CORP.
GIRAGOSIAN, PAUL	THE MITRE CORP.
GLASS, ROBERT L	COMPUTING TRENDS
GODFREY, PARKE	UNIVERSITY OF MARYLAND
GODFREY, SALLY	NASA/GSFC
GOGIA, B. K	ENGINEERING & ECONOMY RESEARCH, INC.
GOINS, MELVIN	DEPT. OF DEFENSE
GOLDEN, JAMES H	SANDERS ASSOCIATION
GOLDEN, JOHN R	EASTMAN KODAK CO.
GOLDSMITH, LARRY	DEPT. OF LABOR
GORDON, HAYDEN H	COMPUTER SCIENCES CORP.
GORDON, MARC D	BOOZ, ALLEN & HAMILTON, INC.
GOUW, ROBERT	TRW
GRAHAM, MARCELLUS	NASA/MSFC
GRAVES, RUSSELL J	DEPT. OF DEFENSE

FOURTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP ATTENDEES

GRAVITTE, JUNE A.	FORD AEROSPACE CO.
GREEN, DANIEL	U.S. AIR FORCE
GREEN, DAVID	COMPUTER SCIENCES CORP.
GREEN, SCOTT	NASA/GSFC
GREGORY, SAMUEL T.	
GRIMALDI, STEVE	BOOZ, ALLEN & HAMILTON, INC.
GRONDALSKI, JEAN	COMPUTER SCIENCES CORP.
GUENTERBERG, SHARON	PLANNING RESEARCH CORP.
GUPTA, LAKSHMI	FORD AEROSPACE CO.
HALL, DANA	SYSTEMS ENGINEERING AND SECURITY, INC
HALL, GARDINER	FORD AEROSPACE CO.
HALTERMAN, KAREN	NASA/GSFC
HARRIS, ALAN W.	LOGICAN, INC.
HARRIS, BERNARD	NASA/GSFC
HAYES, CAROL	UNISYS CORP.
HEASTY, RICHARD	COMPUTER SCIENCES CORP.
HECHT, MYRON	SOHAR, INC.
HECK, JOANN L.	COMPUTER SCIENCES CORP.
HEFFERNAN, HENRY G.	EDP NEWS SERVICES
HELLER, GERRY	COMPUTER SCIENCES CORP.
HENDRICK, ROBERT B.	COMPUTER SCIENCES CORP.
HENRY-NICKENS, STEPHANIE	NASA/GSFC
HERBOLSHEIMER, CHARLES	FEDERAL AVIATION AGENCY
HILL, KEN	NASA/GSFC
HILL, MIKE	MARTIN MARIETTA
HIOTT, JIM	UNISYS CORP.
HOCHHAUSER, S.	SOHAR, INC.
HODGES, DEIDRA	MARTIN MARIETTA
HOLLADAY, WENDY T.	NASA
HOLMES, BARBARA	CRMI
HOOTEN, MONICA	FORD AEROSPACE CO.
HORMBY, TOM W.	JOHNS HOPKINS UNIVERSITY
HOUSTON, SUSAN	LISAN CORP.
HUMPHREY, WATTS	SOFTWARE ENGINEERING INSTITUTE
IDELSON, NORMAN	ARINC RESEARCH CORP.
IRELAND, THOMAS	TEKTRONIX DEFENSE SYSTEMS
ISKOW, LARRY	CENSUS BUREAU
JAHANGIRI, MAJID	COMPUTER SCIENCES CORP.
JAKAITIS, JOYCE	AMERICAN SYSTEMS CORP.
JELETIC, JIM	NASA/GSFC
JENKINS-BNAFA, JOVITA	TRW
JOESTING, DAVID	BENDIX FIELD ENGINEERING CORP.
JOHANNSON, HANK	FORD AEROSPACE CO.
JONES, CARL	SCIENCE APPLICATIONS, INC.
JONES, DAVID	UNISYS CORP.
JORDAN, LEON	COMPUTER SCIENCES CORP.
KARDATZKE, OWEN	NASA/GSFC
KARLIN, JAY	PROJECT ENGINEERING, INC.
KEARNEY, ROBERT	PLANNING RESEARCH CORP.

FOURTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP ATTENDEES

KELLY, JOHN C.	JET PROPULSION LAB
KELLY, KIM R.	IBM
KENNEDY, ELIZABETH A.	ROCKWELL INTERNATIONAL
KESTER, RUSH	COMPUTER SCIENCES CORP.
KHAITAN, ANURAG	UNIVERSITY OF MARYLAND
KICKLIGHTER, BOB	NATIONAL LIBRARY OF MEDICINE
KILE, THOMAS	DEPT. OF THE ARMY
KIMMINAU, PAMELA S.	DEPT. OF DEFENSE
KIRKPATRICK, MARK	CARLOW ASSOC.
KISHAN, SUSHMA	STANFORD TELECOMMUNICATIONS, INC.
KLEMM, DANIEL	FORD AEROSPACE CO.
KNIGHT, JOHN C.	UNIVERSITY OF VIRGINIA
KOESER, KEN	VITRO CORP.
KOPP, ALLAN	TELESOFT
KOUCHAKDJIAN, ARA	UNIVERSITY OF MARYLAND
KRAHN, MARGIE	DEPT. OF DEFENSE
KRALY, KAREN	NATIONAL LIBRARY OF MEDICINE
KRAMER, NANCY	PLANNING RESEARCH CORP.
KRAUS, PAUL J.	COMPUTATIONAL ENGINEERING, INC.
KRIEGMAN, DAVID	SRA CORP.
KUDLINSKI, ROBERT A.	NASA/LARC
KUHN, RICK	NATIONAL BUREAU OF STANDARDS
KUNKEL, HENRY	BOEING AEROSPACE CO.
LABAUGH, ROBERT	MARTIN MARIETTA
LAL, NAND	NASA/GSFC
LAMAS, NIKI	CENSUS BUREAU
LANDIS, LINDA	COMPUTER SCIENCES CORP.
LAVALLEE, DAVID	FORD AEROSPACE CO.
LEAKE, STEPHEN	NIST
LEE, JOHN A.	GENERAL DYNAMICS
LEHMAN, MANNY	IMPERIAL COLLEGE
LEVAY, KAREN	COMPUTER SCIENCES CORP.
LEVESON, NANCY G.	UNIVERSITY OF CALIFORNIA
LEVITT, DAVID S.	COMPUTER SCIENCES CORP.
LIGHT, WARREN	CTA, INC.
LIN, CHI Y.	JET PROPULSION LAB
LITTLEWOOD, CHRISTOPHER	MARTIN MARIETTA
LIU, JEAN C.	COMPUTER SCIENCES CORP.
LIU, KUEN-SAN	COMPUTER SCIENCES CORP.
LOCKMAN, ABE	GTE
LOESH, BOB E.	JET PROPULSION LAB
LOTT, CHRIS	UNIVERSITY OF MARYLAND
LUCIER, ERNIE	NASA/HEADQUARTERS
LUCZAK, RAY	COMPUTER SCIENCES CORP.
LYTTON, VICTOR H.	DEPT. OF AGRICULTURE
LYU, MICHAEL	BRONX COMMUNITY COLLEGE
LYU, MICHAEL R.	JET PROPULSION LAB
LaMARSH, MARGO	NASA/LARC
MACCHINI, BRUNO	UNIVERSITY OF MARYLAND
MADDOCK, KAREN R.	TECHNOLOGY PLANNING, INC.
MADSEN, KENT	UNIVERSITY OF CALIFORNIA

FOURTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP ATTENDEES

MAGILL, ELEANORE L.	GENERAL ELECTRIC CORP.
MALAY, SUSAN	PLANNING ANALYSIS CORP.
MALLET, BOB	TECHNOLOGY PLANNING, INC.
MALTHOUSE, NANCY	LOGICON, INC.
MARKS, TOM	DEPT. OF DEFENSE
MARSHLICK, MICHAEL	COMPUTER SCIENCES CORP.
MARTINEZ, BILL	FORD AEROSPACE CO.
MARTSCHENKO, WILLIAM N.	UNIVERSITY OF MARYLAND
MARVRAY, ESMOND	NASA/GSFC
MATHIASEN, CANDY	UNISYS CORP.
MCCLURE, MARTY	BENDIX FIELD ENGINEERING CORP.
MCCOMAS, DAVID	NASA/GSFC
MCDERMOTT, TIM	COMPUTER SCIENCES CORP.
MCDONALD, BETH	DEPT. OF DEFENSE
MCGARRY, FRANK	NASA/GSFC
MCGARRY, PETER	GENERAL ELECTRIC CORP.
MCGOWAN, CLEMENT	CONTEL TECHNOLOGY CENTER
MCKENNA, JOHN J.	DEPT. OF DEFENSE
MCWEE, HARRY	DEPT. OF DEFENSE
MEHLER, STEVE	IIT RESEARCH INSTITUTE
MERIFIELD, JAMES	ADVANCED TECHNOLOGY, INC.
MICKEL, SUSAN	GENERAL ELECTRIC CORP.
MISHOE, JAMES P.	IIT RESEARCH INSTITUTE
MOLESKI, LAURA	CRMI
MOLESKI, WALT	NASA/GSFC
MOONEY, PAT	IBM
MORUSIEWICZ, LINDA M.	COMPUTER SCIENCES CORP.
MOYLEN, ALDEN	COMPUTER SCIENCES CORP.
MUDRONE, JAMES	DEPT. OF DEFENSE
MULLER, ERICH	SPARTA, INC.
MUSA, JOHN D.	AT&T BELL LABS
MYERS, MONTGOMERY	UNISYS CORP.
MYERS, PHILIP I.	COMPUTER SCIENCES CORP.
NARROWS, BERNIE	BENDIX FIELD ENGINEERING CORP.
NICKENS, DON O.	HARRIS SPACE SYSTEMS CORP.
NORCIO, TONY F.	UNIVERSITY OF MARYLAND
NORO, MASAMI	UNIVERSITY OF MARYLAND
O'BRIEN, DAVID	CONCURRENT COMPUTER CO.
O'BRIEN, ROBERT	NASA/GSFC
O'MALLEY, JAMES	HGO TECHNOLOGY
O'MALLEY, RUTH E.	HGO TECHNOLOGY
OHLMACHER, JANE	SOCIAL SECURITY ADMINISTRATION
PAGE, GERALD	COMPUTER SCIENCES CORP.
PAJERSKI, ROSE	NASA/GSFC
PEARSON, BOYD	NASA/GSFC
PELNIK, TAMMY M.	THE MITRE CORP.
PENNEY, LEONIE	PENNEY ASSOCIATES
PEREZ, FRANK	UNISYS CORP.
PERKINS, DOROTHY	NASA/GSFC
PETERSEN, JANE B.	AUTOMETRIC, INC.

FOURTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP ATTENDEES

PFLARTER, DAVE	MCDONNELL DOUGLAS CORP.
PIETRASANTA, AL	JET PROPULSION LAB
PLETT, MICHAEL E.	COMPUTER SCIENCES CORP.
PLUNKETT, THERESA	DEPT. OF DEFENSE
POLE, THOMAS	SOFTWARE PRODUCTIVITY CONSORTIUM
POLLACK, JAY	COMPUTER SCIENCES CORP.
PORTER, ADAM A.	UNIVERSITY OF CALIFORNIA
POTTER, WILLIAM	NASA/GSFC
PRESSMAN, TOM	STRICTLY BUSINESS COMPUTER SYSTEMS
PRINCE, ANDY	PLANNING RESEARCH CORP.
PRISEKIN, JULIA	IIT RESEARCH INSTITUTE
PUGH, DOUGLAS H.	IIT RESEARCH INSTITUTE
PUMPHREY, KAREN	COMPUTER SCIENCES CORP.
PURCELL, ELIZABETH	THE MITRE CORP.
PUTNEY, BARBARA	NASA/GSFC
QUANN, EILEEN S.	FASTRAK TRAINING, INC.
RADOSEVICH, JIM	NASA/HEADQUARTERS
RANADE, PRAKASH V.	COMPUTER SCIENCES CORP.
RANEY, DALE L.	UNISYS CORP.
RAPP, DAVE	DEPT. OF DEFENSE
REDDING, JOHN	DEFENSE COMMUNICATIONS AGENCY
REDDY, K G.	VNG SOFTWARE CONSULTING SERVICES
RICHARD, DAN	IBM
RITTER, SHEILA J.	NASA/GSFC
ROBILLARD, PIERRE N.	UNIVERSITY OF MONTREAL
ROBINSON, ALICE B.	NASA/HEADQUARTERS
ROBINSON, STEVE	DYNAMICS RESEARCH CORP.
RODA, A. C.	PLANNING RESEARCH CORP.
ROGERS, KATHY	THE MITRE CORP.
ROMBACH, DIETER H.	UNIVERSITY OF MARYLAND
ROTTERMAN, GENE	GENERAL DYNAMICS
ROY, DAN	FORD AEROSPACE CO.
RUDOLPH, RUTH	COMPUTER SCIENCES CORP.
SALASIN, JOHN	GTE
SANDERS, ANTONIO	NASA/GSFC
SARY, CHARISSE	COMPUTER SCIENCES CORP.
SAUBLE, GEORGE	OMITRON, INC.
SAVANH, VIRASACH	DEPT. OF LABOR
SCHEIDT, DAVE	IIT RESEARCH INSTITUTE
SCHELLHASE, RONALD J.	COMPUTER SCIENCES CORP.
SCHMIDT, SANDY	BOOZ, ALLEN & HAMILTON, INC.
SCHULER, MARY P.	NASA/LARC
SCHWARTZ, BENJAMIN L.	THE ANALYTIC SCIENCES CORP.
SCOTT, STEVE	UNISYS CORP.
SEAVER, DAVID P.	PROJECT ENGINEERING, INC.
SEIDEWITZ, ED	NASA/GSFC
SELBY, RICHARD W.	UNIVERSITY OF CALIFORNIA AT IRVINE
SEVERINO, TONY	GENERAL ELECTRIC/RCA
SHANKLIN, ROBERT	COMPUTER SCIENCES CORP.
SHAWE, M.	BENDIX FIELD ENGINEERING CORP.

FOURTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP ATTENDEES

SHEKARCHI, JOHN	COMPUTER SCIENCES CORP.
SHEPPARD, SYLVIA B.	NASA/GSFC
SHOAN, WENDY	NASA/GSFC
SIEGERT, GREG	IIT RESEARCH INSTITUTE
SILBERBERG, DAVID	NATIONAL COMPUTER SECURITY CENTER
SLACK, IKE	MCDONNELL DOUGLAS ASTRONAUTICS CO.
SLEDGE, FRANK	GTE
SLOVIN, MALCOLM	COMPUTER SCIENCES CORP.
SMITH, CASSANDRA	THE MITRE CORP.
SMITH, GENE	NASA/GSFC
SMITH, LAURIE	COMPUTER SCIENCES CORP.
SMITH, M C.	THE MITRE CORP.
SMITH, OLIVER	EG&G WASC, INC.
SMITH, PAUL H.	NASA/HEADQUARTERS
SO, MARIA	MCDONNELL DOUGLAS SYSTEMS CORP.
SOL-GUTIERREZ, ANA	FORD AEROSPACE CO.
SOLOMON, CARL	ST SYSTEMS CORP.
SORKOWITZ, AL R.	DEPT. OF THE NAVY
SOVA, DON	NASA/HEADQUARTERS
SPANGLER, ALAN	IBM
SPENCE, BAILEY	COMPUTER SCIENCES CORP.
SPIEGEL, DOUG	NASA/GSFC
SQUIRE, JON	WESTINGHOUSE ELECTRIC CORP.
SQUIRES, BURTON E.	CONSULTANT
STAFFORD, BRUCE	IRS
STALLARD, JOHN	DEFENSE COMMUNICATIONS AGENCY
STARK, MICHAEL	NASA/GSFC
STEGER, WARREN	COMPUTER SCIENCES CORP.
STEINBACHER, JODY	NASA/JPL
STICKLE, RICHARD	HEI
STOKES, ED	COMPUTER SCIENCES CORP.
STRAUB, PABLO	UNIVERSITY OF MARYLAND
STUART, ANTOINETTE D.	DEPT. OF THE NAVY
SUD, VED	THE MITRE CORP.
SUN, ALICE	THE MITRE CORP.
SWALTZ, LEON	IBM
SZULEWSKI, PAUL	C. S. DRAPER LAB, INC.
TANG, Y. K.	FORD AEROSPACE CO.
TASAKI, KEIJI	NASA/GSFC
TAUSWORTHE, BOB	NASA/JPL
TAVASSOLI, NAZ	COMPUTER SCIENCES CORP.
TAYLOR, GUY	FLEET COMBAT DIRECTION SYSTEMS
THACKREY, KENT	PLANNING ANALYSIS CORP.
THOMAS, DONNA	COMPUTER SCIENCES CORP.
THOMPSON, JOHN T.	FORD AEROSPACE CO.
THORNTON, THOMAS	NASA/JPL
THREADGILL, PETER	DEPT. OF DEFENSE
TIAN, JIANHUI	UNIVERSITY OF MARYLAND
TRAYSYELUE, WEISNER	COMPUTER SCIENCES CORP.
TRUSZKOWSKI, WALT F.	NASA/GSFC
TZENG, NIGL	NASA/STX

FOURTEENTH ANNUAL SOFTWARE ENGINEERING WORKSHOP ATTENDEES

ULERY, BRADFORD	UNIVERSITY OF MARYLAND
ULLMAN, RICHARD	ST SYSTEMS CORP.
URBINA, DANIEL	FORD AEROSPACE CO.
URR, CLIFFORD	PLANNING ANALYSIS CORP.
VALETT, JON	NASA/GSFC
VALETT, SUSAN	NASA/GSFC
VANDERGRAFT, JAMES S	COMPUTATIONAL ENGINEERING, INC.
VAUGHAN, JOE	SOCIAL SECURITY ADMINISTRATION
VEHMEIER, DAWN R	OASD(P&L)WSIG
VERNACCHIO, AL	NASA/GSFC
VIENNEAU, ROBERT	KAMAN SCIENCES CORP.
VOIGT, DAVID	BENDIX FIELD ENGINEERING CORP.
VOIGT, SUSAN	NASA/LARC
VUOLO, BOB	NASA/JPL
WALIGORA, SHARON R	COMPUTER SCIENCES CORP.
WALKER, GARY N	JET PROPULSION LAB
WALKER, JOHN	IIT RESEARCH INSTITUTE
WALL, TIM	SPARTA, INC.
WALLACE, DOLORES	NATIONAL INSTITUTE OF STANDARDS & TEC
WARTIK, STEVEN	SOFTWARE PRODUCTIVITY CONSORTIUM
WATSON, BARRY	IIT RESEARCH INSTITUTE
WAUGH, DOUG	IBM
WEBSTER, THOMAS M	COMPUTATIONAL ENGINEERING, INC.
WEEKLEY, JIM	FORD AEROSPACE CO.
WEISMAN, DAVID	UNISYS CORP.
WEISS, DAVE	SOFTWARE PRODUCTIVITY CONSORTIUM
WENDE, CHARLES	NASA/GSFC
WENDE, ROY	FAIRCHILD SPACE CO.
WESTON, WILLIAM	NASA/GSFC
WHITESELL, STEVEN A	COMPUTER SCIENCES CORP.
WILBERT, CARL K	NASA/HEADQUARTERS
WILDER, DAVID C	DEPT. OF DEFENSE
WILLIAMS, CHERYL	CTA, INC.
WILSON, BILL M	QUONG ASSOC.
WILSON, RUSSELL	BOEING AEROSPACE CO.
WITTIG, MIKE	IIT RESEARCH INSTITUTE
WONG, ALICE A	FEDERAL AVIATION AGENCY
WONG, WILLIAM	NATIONAL INSTITUTE OF STANDARDS & TEC
WOOD, DICK	COMPUTER SCIENCES CORP.
WOOD, TERRI	NASA/GSFC
YANG, CHAO	NASA/GSFC
YOUMAN, CHARLES	THE MITRE CORP.
ZAVELER, SAUL	U.S. AIR FORCE
ZAWILSKI, TONY	THE MITRE CORP.
ZELKOWITZ, MARV	UNIVERSITY OF MARYLAND
ZIMET, BETH	COMPUTER SCIENCES CORP.
ZIMMER, JANET	IIT RESEARCH INSTITUTE

APPENDIX B — SEL BIBLIOGRAPHY

STANDARD BIBLIOGRAPHY OF SEL LITERATURE

The technical papers, memorandums, and documents listed in this bibliography are organized into two groups. The first group is composed of documents issued by the Software Engineering Laboratory (SEL) during its research and development activities. The second group includes materials that were published elsewhere but pertain to SEL activities.

SEL-ORIGINATED DOCUMENTS

SEL-76-001, Proceedings From the First Summer Software Engineering Workshop, August 1976

SEL-77-002, Proceedings From the Second Summer Software Engineering Workshop, September 1977

SEL-77-004, A Demonstration of AXES for NAVPAK, M. Hamilton and S. Zeldin, September 1977

SEL-77-005, GSFC NAVPAK Design Specifications Languages Study, P. A. Scheffer and C. E. Velez, October 1977

SEL-78-005, Proceedings From the Third Summer Software Engineering Workshop, September 1978

SEL-78-006, GSFC Software Engineering Research Requirements Analysis Study, P. A. Scheffer and C. E. Velez, November 1978

SEL-78-007, Applicability of the Rayleigh Curve to the SEL Environment, T. E. Mapp, December 1978

SEL-78-302, FORTRAN Static Source Code Analyzer Program (SAP) User's Guide (Revision 3), W. J. Decker and W. A. Taylor, July 1986

SEL-79-002, The Software Engineering Laboratory: Relationship Equations, K. Freburger and V. R. Basili, May 1979

SEL-79-003, Common Software Module Repository (CSMR) System Description and User's Guide, C. E. Goorevich, A. L. Green, and S. R. Waligora, August 1979

SEL-79-004, Evaluation of the Caine, Farber, and Gordon Program Design Language (PDL) in the Goddard Space Flight Center (GSFC) Code 580 Software Design Environment, C. E. Goorevich, A. L. Green, and W. J. Decker, September 1979

SEL-79-005, Proceedings From the Fourth Summer Software Engineering Workshop, November 1979

SEL-80-002, Multi-Level Expression Design Language-Requirement Level (MEDL-R) System Evaluation, W. J. Decker and C. E. Goorevich, May 1980

SEL-80-003, Multimission Modular Spacecraft Ground Support Software System (MMS/GSSS) State-of-the-Art Computer Systems/ Compatibility Study, T. Welden, M. McClellan, and P. Liebertz, May 1980

SEL-80-005, A Study of the Musa Reliability Model, A. M. Miller, November 1980

SEL-80-006, Proceedings From the Fifth Annual Software Engineering Workshop, November 1980

SEL-80-007, An Appraisal of Selected Cost/Resource Estimation Models for Software Systems, J. F. Cook and F. E. McGarry, December 1980

SEL-81-008, Cost and Reliability Estimation Models (CAREM) User's Guide, J. F. Cook and E. Edwards, February 1981

SEL-81-009, Software Engineering Laboratory Programmer Workbench Phase 1 Evaluation, W. J. Decker and F. E. McGarry, March 1981

SEL-81-011, Evaluating Software Development by Analysis of Change Data, D. M. Weiss, November 1981

SEL-81-012, The Rayleigh Curve as a Model for Effort Distribution Over the Life of Medium Scale Software Systems, G. O. Picasso, December 1981

SEL-81-013, Proceedings From the Sixth Annual Software Engineering Workshop, December 1981

SEL-81-014, Automated Collection of Software Engineering Data in the Software Engineering Laboratory (SEL), A. L. Green, W. J. Decker, and F. E. McGarry, September 1981

SEL-81-101, Guide to Data Collection, V. E. Church, D. N. Card, F. E. McGarry, et al., August 1982

SEL-81-104, The Software Engineering Laboratory, D. N. Card, F. E. McGarry, G. Page, et al., February 1982

SEL-81-107, Software Engineering Laboratory (SEL) Compendium of Tools, W. J. Decker, W. A. Taylor, and E. J. Smith, February 1982

SEL-81-110, Evaluation of an Independent Verification and Validation (IV&V) Methodology for Flight Dynamics, G. Page, F. E. McGarry, and D. N. Card, June 1985

SEL-81-205, Recommended Approach to Software Development, F. E. McGarry, G. Page, S. Eslinger, et al., April 1983

SEL-82-001, Evaluation of Management Measures of Software Development, G. Page, D. N. Card, and F. E. McGarry, September 1982, vols. 1 and 2

SEL-82-004, Collected Software Engineering Papers: Volume 1, July 1982

SEL-82-007, Proceedings From the Seventh Annual Software Engineering Workshop, December 1982

SEL-82-008, Evaluating Software Development by Analysis of Changes: The Data From the Software Engineering Laboratory, V. R. Basili and D. M. Weiss, December 1982

SEL-82-102, FORTRAN Static Source Code Analyzer Program (SAP) System Description (Revision 1), W. A. Taylor and W. J. Decker, April 1985

SEL-82-105, Glossary of Software Engineering Laboratory Terms, T. A. Babst, F. E. McGarry, and M. G. Rohleder, October 1983

SEL-82-806, Annotated Bibliography of Software Engineering Laboratory Literature, M. Buhler and J. Valett, November 1989

SEL-83-001, An Approach to Software Cost Estimation, F. E. McGarry, G. Page, D. N. Card, et al., February 1984

SEL-83-002, Measures and Metrics for Software Development, D. N. Card, F. E. McGarry, G. Page, et al., March 1984

SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983

SEL-83-006, Monitoring Software Development Through Dynamic Variables, C. W. Doerflinger, November 1983

SEL-83-007, Proceedings From the Eighth Annual Software Engineering Workshop, November 1983

SEL-83-106, Monitoring Software Development Through Dynamic Variables (Revision 1), C. W. Doerflinger, November 1989

SEL-84-001, Manager's Handbook for Software Development, W. W. Agresti, F. E. McGarry, D. N. Card, et al., April 1984

SEL-84-003, Investigation of Specification Measures for the Software Engineering Laboratory (SEL), W. W. Agresti, V. E. Church, and F. E. McGarry, December 1984

SEL-84-004, Proceedings From the Ninth Annual Software Engineering Workshop, November 1984

SEL-85-001, A Comparison of Software Verification Techniques, D. N. Card, R. W. Selby, Jr., F. E. McGarry, et al., April 1985

SEL-85-002, Ada Training Evaluation and Recommendations From the Gamma Ray Observatory Ada Development Team, R. Murphy and M. Stark, October 1985

SEL-85-003, Collected Software Engineering Papers: Volume III, November 1985

SEL-85-004, Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics, R. W. Selby, Jr., May 1985

SEL-85-005, Software Verification and Testing, D. N. Card, C. Antle, and E. Edwards, December 1985

SEL-85-006, Proceedings From the Tenth Annual Software Engineering Workshop, December 1985

SEL-86-001, Programmer's Handbook for Flight Dynamics Software Development, R. Wood and E. Edwards, March 1986

SEL-86-002, General Object-Oriented Software Development, E. Seidewitz and M. Stark, August 1986

SEL-86-003, Flight Dynamics System Software Development Environment Tutorial, J. Buell and P. Myers, July 1986

SEL-86-004, Collected Software Engineering Papers: Volume IV, November 1986

SEL-86-005, Measuring Software Design, D. N. Card, October 1986

SEL-86-006, Proceedings From the Eleventh Annual Software Engineering Workshop, December 1986

SEL-87-001, Product Assurance Policies and Procedures for Flight Dynamics Software Development, S. Perry et al., March 1987

SEL-87-002, Ada Style Guide (Version 1.1), E. Seidewitz et al., May 1987

SEL-87-003, Guidelines for Applying the Composite Specification Model (CSM), W. W. Agresti, June 1987

SEL-87-004, Assessing the Ada Design Process and Its Implications: A Case Study, S. Godfrey, C. Brophy, et al., July 1987

SEL-87-008, Data Collection Procedures for the Rehosted SEL Database, G. Heller, October 1987

SEL-87-009, Collected Software Engineering Papers: Volume V, S. DeLong, November 1987

SEL-87-010, Proceedings From the Twelfth Annual Software Engineering Workshop, December 1987

SEL-88-001, System Testing of a Production Ada Project: The GRODY Study, J. Seigle, L. Esker, and Y. Shi, November 1988

SEL-88-002, Collected Software Engineering Papers: Volume VI, November 1988

SEL-88-003, Evolution of Ada Technology in the Flight Dynamics Area: Design Phase Analysis, K. Quimby and L. Esker, December 1988

SEL-88-004, Proceeding of the Thirteenth Annual Software Engineering Workshop, November 1988

SEL-88-005, Proceedings of the First NASA Ada User's Symposium, December 1988

SEL-89-002, Implementation of a Production Ada Project: The GRODY Study, S. Godfrey and C. Brophy, September 1989

SEL-89-003, Software Management Environment (SME) Concepts and Architecture, W. Decker and J. Valett, August 1989

SEL-89-004, Evolution of Ada Technology in the Flight Dynamics Area: Implementation/Testing Phase Analysis, K. Quimby, L. Esker, L. Smith, M. Stark, and F. McGarry, November 1989

SEL-89-005, Lessons Learned in the Transition to Ada From FORTRAN at NASA/Goddard, C. Brophy, November 1989

SEL-89-006, Collected Software Engineering Papers: Volume VII, November 1989

SEL-89-007, Proceedings of the Fourteenth Annual Software Engineering Workshop, November 1989

SEL-89-008, Proceedings of the Second NASA Ada Users' Symposium, November 1989

SEL-89-101, Software Engineering Laboratory (SEL) Database Organization and User's Guide (Revision 1), M. So, G. Heller, S. Steinberg, K. Pumphrey, and D. Spiegel, February 1990

SEL-90-001, Database Access Manager for the Software Engineering Laboratory (DAMSEL) User's Guide, M. Buhler and K. Pumphrey, April 1990

SEL-RELATED LITERATURE

[4]Agresti, W. W., V. E. Church, D. N. Card, and P. L. Lo, "Designing With Ada for Satellite Simulation: A Case Study," Proceedings of the First International Symposium on Ada for the NASA Space Station, June 1986

[2]Agresti, W. W., F. E. McGarry, D. N. Card, et al., "Measuring Software Technology," Program Transformation and Programming Environments. New York: Springer-Verlag, 1984

[1]Bailey, J. W., and V. R. Basili, "A Meta-Model for Software Development Resource Expenditures," Proceedings of the Fifth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1981

[7]Basili, V. R., Maintenance = Reuse-Oriented Software Development, University of Maryland, Technical Report TR-2244, May 1989

[1]Basili, V. R., "Models and Metrics for Software Management and Engineering," ASME Advances in Computer Technology, January 1980, vol. 1

[7]Basili, V. R., Software Development: A Paradigm for the Future, University of Maryland, Technical Report TR-2263, June 1989

Basili, V. R., Tutorial on Models and Metrics for Software Management and Engineering. New York: IEEE Computer Society Press, 1980 (also designated SEL-80-008)

[3]Basili, V. R., "Quantitative Evaluation of Software Methodology," Proceedings of the First Pan-Pacific Computer Conference, September 1985

[1]Basili, V. R., and J. Beane, "Can the Parr Curve Help With Manpower Distribution and Resource Estimation Problems?," Journal of Systems and Software, February 1981, vol. 2, no. 1

[1]Basili, V. R., and K. Freburger, "Programming Measurement and Estimation in the Software Engineering Laboratory," Journal of Systems and Software, February 1981, vol. 2, no. 1

[3]Basili, V. R., and N. M. Panlilio-Yap, "Finding Relationships Between Effort and Other Variables in the SEL," Proceedings of the International Computer Software and Applications Conference, October 1985

[4]Basili, V. R., and D. Patnaik, A Study on Fault Prediction and Reliability Assessment in the SEL Environment, University of Maryland, Technical Report TR-1699, August 1986

[2]Basili, V. R., and B. T. Perricone, "Software Errors and Complexity: An Empirical Investigation," Communications of the ACM, January 1984, vol. 27, no. 1

[1]Basili, V. R., and T. Phillips, "Evaluating and Comparing Software Metrics in the Software Engineering Laboratory," Proceedings of the ACM SIGMETRICS Symposium/Workshop: Quality Metrics, March 1981

Basili, V. R., and J. Ramsey, Structural Coverage of Functional Testing, University of Maryland, Technical Report TR-1442, September 1984

[3]Basili, V. R., and C. L. Ramsey, "ARROWSMITH-P--A Prototype Expert System for Software Engineering Management," Proceedings of the IEEE/MITRE Expert Systems in Government Symposium, October 1985

Basili, V. R., and R. Reiter, "Evaluating Automatable Measures for Software Development," Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost. New York: IEEE Computer Society Press, 1979

[5]Basili, V., and H. D. Rombach, "Tailoring the Software Process to Project Goals and Environments," Proceedings of the 9th International Conference on Software Engineering, March 1987

[5]Basili, V., and H. D. Rombach, "T A M E: Tailoring an Ada Measurement Environment," Proceedings of the Joint Ada Conference, March 1987

[5]Basili, V., and H. D. Rombach, "T A M E: Integrating Measurement Into Software Environments," University of Maryland, Technical Report TR-1764, June 1987

[6]Basili, V. R., and H. D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," IEEE Transactions on Software Engineering, June 1988

[7]Basili, V. R., and H. D. Rombach, Towards A Comprehensive Framework for Reuse: A Reuse-Enabling Software Evolution Environment, University of Maryland, Technical Report TR-2158, December 1988

[2]Basili, V. R., R. W. Selby, Jr., and T. Phillips, "Metric Analysis and Data Validation Across FORTRAN Projects," IEEE Transactions on Software Engineering, November 1983

[3]Basili, V. R., and R. W. Selby, Jr., "Calculation and Use of an Environments's Characteristic Software Metric Set," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

Basili, V. R., and R. W. Selby, Jr., Comparing the Effectiveness of Software Testing Strategies, University of Maryland, Technical Report TR-1501, May 1985

[3]Basili, V. R., and R. W. Selby, Jr., "Four Applications of a Software Data Collection and Analysis Methodology," Proceedings of the NATO Advanced Study Institute, August 1985

[4]Basili, V. R., R. W. Selby, Jr., and D. H. Hutchens, "Experimentation in Software Engineering," IEEE Transactions on Software Engineering, July 1986

[5]Basili, V. and R. Selby, Jr., "Comparing the Effectiveness of Software Testing Strategies," IEEE Transactions on Software Engineering, December 1987

[2]Basili, V. R., and D. M. Weiss, A Methodology for Collecting Valid Software Engineering Data, University of Maryland, Technical Report TR-1235, December 1982

[3]Basili, V. R., and D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data," IEEE Transactions on Software Engineering, November 1984

[1]Basili, V. R., and M. V. Zelkowitz, "The Software Engineering Laboratory: Objectives," Proceedings of the Fifteenth Annual Conference on Computer Personnel Research, August 1977

Basili, V. R., and M. V. Zelkowitz, "Designing a Software Measurement Experiment," Proceedings of the Software Life Cycle Management Workshop, September 1977

[1]Basili, V. R., and M. V. Zelkowitz, "Operation of the Software Engineering Laboratory," Proceedings of the Second Software Life Cycle Management Workshop, August 1978

[1]Basili, V. R., and M. V. Zelkowitz, "Measuring Software Development Characteristics in the Local Environment," Computers and Structures, August 1978, vol. 10

Basili, V. R., and M. V. Zelkowitz, "Analyzing Medium Scale Software Development," Proceedings of the Third International Conference on Software Engineering. New York: IEEE Computer Society Press, 1978

[5]Brophy, C., W. Agresti, and V. Basili, "Lessons Learned in Use of Ada-Oriented Design Methods," Proceedings of the Joint Ada Conference, March 1987

[6]Brophy, C. E., S. Godfrey, W. W. Agresti, and V. R. Basili, "Lessons Learned in the Implementation Phase of a Large Ada Project," Proceedings of the Washington Ada Technical Conference, March 1988

[2]Card, D. N., "Early Estimation of Resource Expenditures and Program Size," Computer Sciences Corporation, Technical Memorandum, June 1982

[2]Card, D. N., "Comparison of Regression Modeling Techniques for Resource Estimation," Computer Sciences Corporation, Technical Memorandum, November 1982

[3]Card, D. N., "A Software Technology Evaluation Program," Annais do XVIII Congresso Nacional de Informatica, October 1985

[5]Card, D., and W. Agresti, "Resolving the Software Science Anomaly," The Journal of Systems and Software, 1987

[6]Card, D. N., and W. Agresti, "Measuring Software Design Complexity," The Journal of Systems and Software, June 1988

Card, D. N., V. E. Church, W. W. Agresti, and Q. L. Jordan, "A Software Engineering View of Flight Dynamics Analysis System," Parts I and II, Computer Sciences Corporation, Technical Memorandum, February 1984

[4]Card, D. N., V. E. Church, and W. W. Agresti, "An Empirical Study of Software Design Practices," IEEE Transactions on Software Engineering, February 1986

Card, D. N., Q. L. Jordan, and V. E. Church, "Characteristics of FORTRAN Modules," Computer Sciences Corporation, Technical Memorandum, June 1984

[5]Card, D., F. McGarry, and G. Page, "Evaluating Software Engineering Technologies," IEEE Transactions on Software Engineering, July 1987

[3]Card, D. N., G. T. Page, and F. E. McGarry, "Criteria for Software Modularization," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

[1]Chen, E., and M. V. Zelkowitz, "Use of Cluster Analysis To Evaluate Software Engineering Methodologies," Proceedings of the Fifth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1981

[4]Church, V. E., D. N. Card, W. W. Agresti, and Q. L. Jordan, "An Approach for Assessing Software Prototypes," ACM Software Engineering Notes, July 1986

[2]Doerflinger, C. W., and V. R. Basili, "Monitoring Software Development Through Dynamic Variables," Proceedings of the Seventh International Computer Software and Applications Conference. New York: IEEE Computer Society Press, 1983

[5]Doubleday, D., "ASAP: An Ada Static Source Code Analyzer Program," University of Maryland, Technical Report TR-1895, August 1987 (NOTE: 100 pages long)

[6]Godfrey, S., and C. Brophy, "Experiences in the Implementation of a Large Ada Project," Proceedings of the 1988 Washington Ada Symposium, June 1988

Hamilton, M., and S. Zeldin, A Demonstration of AXES for NAVPAK, Higher Order Software, Inc., TR-9, September 1977 (also designated SEL-77-005)

Jeffery, D. R., and V. Basili, Characterizing Resource Data: A Model for Logical Association of Software Data, University of Maryland, Technical Report TR-1848, May 1987

[6]Jeffery, D. R., and V. R. Basili, "Validating the TAME Resource Data Model," Proceedings of the Tenth International Conference on Software Engineering, April 1988

[5]Mark, L., and H. D. Rombach, A Meta Information Base for Software Engineering, University of Maryland, Technical Report TR-1765, July 1987

[6]Mark, L., and H. D. Rombach, "Generating Customized Software Engineering Information Bases From Software Process and Product Specifications," Proceedings of the 22nd Annual Hawaii International Conference on System Sciences, January 1989

[5]McGarry, F., and W. Agresti, "Measuring Ada for Software Development in the Software Engineering Laboratory (SEL)," Proceedings of the 21st Annual Hawaii International Conference on System Sciences, January 1988

[7]McGarry, F., L. Esker, and K. Quimby, "Evolution of Ada Technology in a Production Software Environment," Proceedings of the Sixth Washington Ada Symposium (WADAS), June 1989

[3]McGarry, F. E., J. Valett, and D. Hall, "Measuring the Impact of Computer Resource Quality on the Software Development Process and Product," Proceedings of the Hawaiian International Conference on System Sciences, January 1985

National Aeronautics and Space Administration (NASA), NASA Software Research Technology Workshop (Proceedings), March 1980

[3]Page, G., F. E. McGarry, and D. N. Card, "A Practical Experience With Independent Verification and Validation," Proceedings of the Eighth International Computer Software and Applications Conference, November 1984

[5]Ramsey, C., and V. R. Basili, An Evaluation of Expert Systems for Software Engineering Management, University of Maryland, Technical Report TR-1708, September 1986

[3]Ramsey, J., and V. R. Basili, "Analyzing the Test Process Using Structural Coverage," Proceedings of the Eighth International Conference on Software Engineering. New York: IEEE Computer Society Press, 1985

[5]Rombach, H. D., "A Controlled Experiment on the Impact of Software Structure on Maintainability," IEEE Transactions on Software Engineering, March 1987

[6]Rombach, H. D., and V. R. Basili, "Quantitative Assessment of Maintenance: An Industrial Case Study," Proceedings From the Conference on Software Maintenance, September 1987

[6]Rombach, H. D., and L. Mark, "Software Process and Product Specifications: A Basis for Generating Customized SE Information Bases," Proceedings of the 22nd Annual Hawaii International Conference on System Sciences, January 1989

[7]Rombach, H. D., and B. T. Ulery, Establishing a Measurement Based Maintenance Improvement Program: Lessons Learned in the SEL, University of Maryland, Technical Report TR-2252, May 1989

[5]Seidewitz, E., "General Object-Oriented Software Development: Background and Experience," Proceedings of the 21st Hawaii International Conference on System Sciences, January 1988

[6]Seidewitz, E., "General Object-Oriented Software Development with Ada: A Life Cycle Approach," Proceedings of the CASE Technology Conference, April 1988

[6]Seidewitz, E., "Object-Oriented Programming in Smalltalk and Ada," Proceedings of the 1987 Conference on Object-Oriented Programming Systems, Languages, and Applications, October 1987

[4]Seidewitz, E., and M. Stark, "Towards a General Object-Oriented Software Development Methodology," Proceedings of the First International Symposium on Ada for the NASA Space Station, June 1986

[7]Stark, M. E. and E. W. Booth, "Using Ada to Maximize Verbatim Software Reuse," Proceedings of TRI-Ada 1989, October 1989

Stark, M., and E. Seidewitz, "Towards a General Object-Oriented Ada Lifecycle," Proceedings of the Joint Ada Conference, March 1987

[7]Sunazuka. T., and V. R. Basili, Integrating Automated Support for a Software Management Cycle Into the TAME System, University of Maryland, Technical Report TR-2289, July 1989

Turner, C., and G. Caron, A Comparison of RADC and NASA/SEL Software Development Data, Data and Analysis Center for Software, Special Publication, May 1981

Turner, C., G. Caron, and G. Brement, NASA/SEL Data Compendium, Data and Analysis Center for Software, Special Publication, April 1981

[5]Valett, J., and F. McGarry, "A Summary of Software Measurement Experiences in the Software Engineering Laboratory," Proceedings of the 21st Annual Hawaii International Conference on System Sciences, January 1988

[3]Weiss, D. M., and V. R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data From the Software Engineering Laboratory," IEEE Transactions on Software Engineering, February 1985

[5]Wu, L., V. Basili, and K. Reed, "A Structure Coverage Tool for Ada Software Systems," Proceedings of the Joint Ada Conference, March 1987

[1]Zelkowitz, M. V., "Resource Estimation for Medium Scale Software Projects," Proceedings of the Twelfth Conference on the Interface of Statistics and Computer Science. New York: IEEE Computer Society Press, 1979

[2]Zelkowitz, M. V., "Data Collection and Evaluation for Experimental Computer Science Research," Empirical Foundations for Computer and Information Science (proceedings), November 1982

[6]Zelkowitz, M. V., "The Effectiveness of Software Prototyping: A Case Study," Proceedings of the 26th Annual Technical Symposium of the Washington, D. C., Chapter of the ACM, June 1987

[6]Zelkowitz, M. V., "Resource Utilization During Software Development," Journal of Systems and Software, 1988

Zelkowitz, M. V., and V. R. Basili, "Operational Aspects of a Software Measurement Facility," Proceedings of the Software Life Cycle Management Workshop, September 1977

NOTES:

[1] This article also appears in SEL-82-004, Collected Software Engineering Papers: Volume I, July 1982.

[2] This article also appears in SEL-83-003, Collected Software Engineering Papers: Volume II, November 1983.

[3] This article also appears in SEL-85-003, Collected Software Engineering Papers: Volume III, November 1985.

[4] This article also appears in SEL-86-004, Collected Software Engineering Papers: Volume IV, November 1986.

[5] This article also appears in SEL-87-009, Collected Software Engineering Papers: Volume V, November 1987.

[6] This article also appears in SEL-88-002, Collected Software Engineering Papers: Volume VI, November 1988.

[7] This article also appears in SEL-89-006, Collected Software Engineering Papers: Volume VII, November 1989.